Common Algorithm Building Blocks

Mirek Riedewald

Key Learning Goals

- Why is the “order inversion” design pattern called that way?
- Be able to write the MapReduce and Spark pseudo-code for all the algorithms in this module.

Introduction

- We discuss approaches and algorithms that frequently appear as building blocks of larger data processing pipelines:
  - Managing state at different granularities
  - Per-record computation
  - Content-based data splitting
  - Grouping and aggregation
  - Duplicate removal
  - Random sampling and shuffling
  - Quantiles
  - Top-k

Managing State at Different Granularities

- In previous modules, we worked with task-local state and discussed two types of global state.
- The first type of global state is a mechanism for broadcasting *read-only* data to all tasks of a job.
- The second are global counters and accumulators, which are initialized and read out by the driver, but updated by the tasks of a job.
- We will first review those mechanisms, then introduce the "order inversion" design pattern in MapReduce.

Review: Broadcasting Read-Only State

- In Hadoop MapReduce, we can use the job Context object to share a small number of constants with all tasks. Similarly, in Spark all variables created in the driver are automatically serialized and shipped with the tasks.
- The file cache in MapReduce allows broadcasting of larger read-only state via files. In Spark, this is supported by broadcast variables.
- Note the difference between making a dataset the input of a job vs broadcasting it. The former assigns only a partition of the data to each task, while the latter copies the entire file to all workers.

Review: Global Counters

- Hadoop MapReduce global counters and Spark accumulators offer limited functionality to avoid the complexity of shared-memory programs.
  - They are initialized and read out by the driver program, i.e., there is no parallelism.
  - They are updated in parallel by the tasks of a job. However, the only supported update operation is to add a (positive or negative) value. Addition is commutative and associative; hence it works correctly independent of the order in which the updates are applied. This greatly simplifies distributed consensus.

Other Options for Global State?

- Should we manage larger objects in the application master, e.g., a central priority queue for algorithms such as single-source shortest path (discussed in a future module)? No!
  - The master's memory would become a bottleneck.
  - Updates of state require shared-memory style synchronization.
- How about distributing the large object over many workers?
  - This addresses the memory bottleneck at the master.
  - If the object can be updated by multiple tasks, then there still is a need for distributed consensus.
  - If each task has exclusive access to a partition, then we are back to the per-task shared-nothing-style data management of MapReduce and Spark tasks.

The Order Inversion Design Pattern

- In there anything else in the spectrum between local data exclusively accessed by a single task and global counters/accumulators?
- Not really, but we will now introduce a solution that was proposed for MapReduce to give operations on fine-grained data partitions access to coarser-grained data needed by multiple small partitions. For reasons explained later, this was called "order inversion."

Example

- Consider a crowd-sourcing project where citizen scientists report birds they observe. For simplicity, assume participants report a (species, color) pair every time they see a bird. Our goal is for each species and color to estimate the conditional probability of the color, given the species.
  - The probability of the Northern Cardinal (N.C.) being red is defined as P(color = red | species = N.C.).
- We can estimate such probabilities from big data by counting the appropriate quantities. To estimate P(color = C | species = S), we need:
  - Frequency count f(S), i.e., the number of observations for species S. In statistical terms, this is called a marginal.
  - Frequency count f(S, C), i.e., the number of observations matching both species S and color C. In statistical terms, this is called a joint event.
  - E.g., we estimate P(color = red | species = N.C.) by dividing the number of red Northern Cardinal observations by the total number of Northern Cardinal observations (including all colors), i.e., as **f(N.C., red) / f(N.C.)**
- This analysis is an example for estimating relative frequencies, a common data-mining task. Another problem with the same structure is to compute the normalized word co-occurrence vector for each word in a document collection. It measures for each word, which other words occur frequently near it.

Obvious Solution Using "Stripes"

- Both f(S) and f(S, C) need to count per species. Hence the species presents itself as an obvious choice for intermediate key. Starting with this observation, the MapReduce program "falls into place." For each input record (species S, color C), Map emits the record with species S as the key and color C as the value.
  - To enable a Combiner or in-Mapper combining, Map could instead output (C, 1) as the value.
- The Reduce call for species S counts the number of occurrences of each color to get f(S, C) for each C. At the same time, it keeps track of the marginal f(S).

```
// Note: If no combining is used, Map could emit
// (S, C), i.e., S as the key and C as the value.
map( observation = (species S, color C) )
  emit( S, (C, 1) )
```

```
reduce( S , [(C1, n1), (C2, n2),…] )

  // H maps a color to a count
  init hashMap H
  marginal = 0

  for all (C, n) in input list do
    H[C] += n
    marginal += n

  for all C in H do
    emit( (S, C), H[C] / marginal )
```

Discussion of the Stripe-Based Approach

- Why do we call this a “stripe”-based approach? Think of a table where each row is indexed by a species and each column is indexed by a color. A table cell, indexed by the combination of species S and color C, contains count f(S, C). By choosing the species as the key, Reduce works with an entire row of this table, which pictorially looks like a stripe.
- The Stripe, i.e., table row, turns out to be a great fit for relative frequency computation. Each cell in the Stripe has the color frequency for the species; and the sum of these frequencies equals the total for the species. Hence all the data needed for computing f(S, C)/f(S) for species S is in the corresponding stripe.
- Is there a drawback to this approach? There are indeed two major limitations:
  - What if data structure H in Reduce exceeds memory size? This would not happen for the colors here, but it might for other problems with more columns.
  - The degree of parallelism of the Reducer workload is limited by the number of different species. What if we have more machines than species?

New Attempt Using "Pairs"

- We could address the problems of the Stripe-based approach by splitting the Reducer work into smaller units. Unfortunately, any smaller unit would miss some of the joint events needed for computing f(S), the marginal for the species. To see why, consider a program that uses both species and color as the intermediate key. For input record (species S, color C), Map emits ((S, C), 1).
  - Combiners and in-Mapper combining can be applied.
- For key (S, C), Reduce computes f(S, C), the frequency count of that species-color combination. Virtually no memory is used, and the fine granularity enables up to #species · #colors different Reduce tasks.
- So, does this approach have any drawbacks? Yes!
  - How can it compute the marginal f(S)? The Reduce call needs f(S, color) for all colors for species S.
  - This could be addressed by running another simple MapReduce program to pre-compute f(S). However, this approach seems wasteful for Big Data as it reads the input data twice—once for computing the f(S) and then again for computing the f(S, C).
- Can we get the best of both worlds, i.e., the one-pass efficiency of Stripes and the small memory footprint and more fine-grained work partitioning of Pairs?

Fixing the Pairs-Based Approach, Attempt 1

- Let us try and fix the Pairs-based approach so that it can do all work in a single MapReduce job.
- Notice that if keys (S, C1) and (S, C2) are assigned to different Reducers, then no Reducer has access to all data needed for computing f(S). Hence, we must make sure that *all keys (S, any color) for species S end up in the same Reduce task*.
  - We already know how to achieve this by defining a custom Partitioner that assigns a key (S, C) to a Reducer solely based on S, ignoring the value of C.
- While the custom Partitioner guarantees that all records for species S will be processed in the same Reduce task, there still is a separate Reduce call for each species-color combination.

Challenge Question 1

- How can we compute f(S) when each Reduce call only works with a single color for species S?

Challenge Question 1

- How can we compute f(S) when each Reduce call only works with a single color for species S?

Try to find the answer yourself. This helps you learn and understand the material.

If you are stuck despite trying hard for 15 minutes, look at previous modules and see if anything there seems promising.

Challenge Question 1

- How can we compute f(S) when each Reduce call only works with a single color for species S?

Are you sure you want to move on and see our answers?

Challenge Question 1

- How can we compute f(S) when each Reduce call only works with a single color for species S?

Last chance to turn back and work on your own solution…

Challenge Question 1

- How can we compute f(S) when each Reduce call only works with a single color for species S?

Okay, here we go:

Challenge Question 1: Answers

- How can we compute f(S) when each Reduce call only works with a single color for species S?
  - Possible answer 1: The individual Reduce calls for keys (S, C1), (S, C2), etc could be turned into a single Reduce call for species S by using a grouping comparator. (Review the secondary sort design pattern.)

Challenge Question 1: Answers

- How can we compute f(S) when each Reduce call only works with a single color for species S?
  - Possible answer 1: The individual Reduce calls for keys (S, C1), (S, C2), etc could be turned into a single Reduce call for species S by using a grouping comparator. (Review the secondary sort design pattern.)
  - Possible answer 2: State can be maintained across the different Reduce calls for keys (S, C1), (S, C2), etc to keep track of f(S, C) for all colors C of species S. More precisely, both the marginal and a hashmap H that stores the frequency for each color could be defined at the Reducer class level, letting each Reduce call for (S, C) update them accordingly. (Review the in-mapper combining design pattern.)

Challenge Question 2

- Does either of these approaches (grouping comparator, in-Reducer combining) really give us a better solution than the Stripes approach?

Challenge Question 2

- Does either of these approaches (grouping comparator, in-Reducer combining) really give us a better solution than the Stripes approach?

Same approach as before: Try to find the answer yourself.

Write down your own answer before looking at ours.

Challenge Question 2: Answer

- Does either of these approaches (grouping comparator, in-Reducer combining) really give us a better solution than the Stripes approach?
  - No. These "improved" Pairs-based approaches have the same drawbacks as the Stripes-based approach. By using a custom Partitioner that only considers the species, Reduce task granularity is back at the species level, like for Stripes. Similarly, since f(S) is computed together with the f(S, C) frequencies, all separate color frequencies for species S must be kept until the last record for species S is processed. Hence the memory footprint is not smaller than for Stripes either. Essentially the attempt at improving the Pairs-based approach made it "simulate" the Stripe-based approach!

Fixing the Pairs-Based Approach, Attempt 2

- The failed attempts so far had tried to compute f(S) for species S concurrently with the different f(S, C) for that same species. This forced us to (1) send all Map output records for species S to the same Reducer and (2) keep the counts for all f(S, C) around until the very last record for species S was processed by the Reducer, i.e., the moment f(S) would finally be known.
- If the program had known f(S) from the beginning, it could have been "broadcast" to the Reduce calls for keys (S, C), for all colors C. This is similar to global state, but for a smaller scope—here per individual species.
- How can we achieve this sharing of coarser-grained data across functions working on more fine-grained data?

State Sharing with Global Counters

- The program below assumes that there is a global counter for each species. There are two problems with this idea:
  - All species must be known in advance in the driver, before any task accesses the input. We cannot dynamically create new counters in a task.
  - Tasks cannot read the counter value, therefore Reduce has no access to the marginal.
- With global counters not working out, we need another way to make species counts f(S) available in Reduce.

```
map( observation: (species S, color C) )
  // Update global counter for species S.
  // This counter must have been created
  // beforehand by the driver.
  marginal_S.add(1)

  emit( (S, C), 1 )
```

```
reduce( (S, C), [n1, n2,…] )
  frequency = 0

  for all n in input list do
    frequency += n

  // Here we assume that the counter can be read
  // in a Reduce task.
  emit( (S, C), frequency / marginal_S )
```

Solution Using Order Inversion

- Like Pairs, the program below uses intermediate key (species, color). In addition to ((S, C), 1), Map emits ((S, dummy), 1) for computing f(S). We exploit that Reduce calls in the same Reducer are executed in key order: Defining a key comparator that puts "dummy" ahead of each real color guarantees that reduce((S, dummy),…) will be executed before reduce((S, C),…) for any real color C.
- A custom Partitioner that partitions only on the species ensures that all records for a species are processed in the same Reduce task.
- The Reducer class needs a task-level variable to keep track of marginal f(S). Since the Reduce call for the dummy color happens first, f(S) will be known before the Reduce call f(S, C) for any color C. Since the key comparator sorts by species first, it also guarantees that the right marginal f(S) is available even if multiple species are assigned to the same Reduce task.
- The Reduce task now has a constant (i.e., independent of the number of species and colors) memory footprint. It needs the marginal for a single species and a single counter for the currently processed species-color combination.

```
map( …, observation: (species S, color C) ) {
  emit( (S, dummy), 1 )
  emit( (S, C), 1 )
}

Partitioner: partition by species

Key comparator for (species, color):
- Sort by species first, then by color
- Make sure color "dummy" comes
  before all real colors
```

```
Class Reducer {
  marginal    // per-task state

  reduce( (S, C), [n1, n2,…] ) {
    if C = dummy {              // Compute marginal f(S)
      marginal = 0
      for all n in input list do marginal += n
    } else {                    // Real color: compute f(S, C)
      colorCnt = 0
      for all n in input list do colorCnt += n
      emit( (S, C), colorCnt / marginal )
    }
  }
}
```

Discussion

- How does this new solution compare to the previous attempts?
  - Pro: It reads the input only once, like the Stripes approach.
  - Pro: It requires virtually no heap memory, like the initial Pairs approach.
  - Potential con: Map duplicates every input record, therefore two copies of the input data are transferred from Mappers to Reducers. However, in practice combining can often dramatically reduce this data transfer.
  - Same: Reduce granularity still is limited by the number of species.
- Interestingly, this approach *in principle* supports a finer Reduce-task granularity at the cost of more data replication in the Map phase. The code on the next slide illustrates this idea. By producing k replicas in Map, each for a different dummy color, the keys (S, C1), (S, C2), etc. can be distributed over k different Reduce tasks. The Partitioner must ensure that each of these tasks receives one of the (S, $dummy_i$) keys as well, so that it can compute the marginal.
  - For Big-Data problems, k-fold data duplication can result in poor performance unless combining is very effective.
  - It is not clear how to implement this idea. We will see a simpler solution to the partition-granularity challenge in a future module where synthetic keys are introduced.

Multiple Vertical Partitions (on Color)

```
map( observation: (species S, color C) ) {
  emit( (S, dummy_1), 1 )
  emit( (S, dummy_2), 1 )
  …
  emit( (S, dummy_k), 1 )
  emit( (S, C), 1 )
}
```

Partitioner: partition by species and color, distributing the colors for species S over k Reduce tasks and making sure each of these Reduce tasks receives exactly one of the $dummy_i$ colors for that species.

Key comparator for (species, color):
- Sort by species first
- Make sure color "dummy" comes before all real colors

```
Class Reducer {
  marginal

  reduce( (S, C), [n1, n2,…] ) {

    if C = dummy {
      // Compute marginal f(S)
      marginal = 0
      for all n in input list do
        marginal += n
    } else {
      // Real color, hence compute f(S, C)
      colorCnt = 0
      for all n in input list do
        colorCnt += n

      emit( (S, C), colorCnt / marginal )
    }
  }
}
```

Order Inversion Design Pattern Summary

- This design pattern for computing state at different granularities and for controlling the order in which this state is computed, is called "order inversion."
  - In what sense does it invert order? We can compute f(S) as the sum of the f(S, C), summing over all colors C. Hence the "natural" order of computation would be to obtain all the f(S, C) for species S first and then add them up to obtain f(S). The order-inversion design pattern turns this around by first computing f(S) and then the individual f(S, C).
- Without order inversion, the programmer would need either (1) larger and more complex data structures to bring the right data together (e.g., hashmap H for a Stripe) or (2) more MapReduce jobs to compute the intermediate results.
- Order inversion turns a synchronization problem into an ordering problem:
  - MapReduce gives the programmer no explicit synchronization primitives. Hence order inversion relies on the key order to enforce a computation order.
- Tradeoff: This approach enables the use of simpler data structures and less Reducer memory, without requiring additional MapReduce phases. It can achieve more fine-grained partitioning at the cost of data replication. A Combiner or in-Mapper combining may soften the negative performance impact of this data replication.

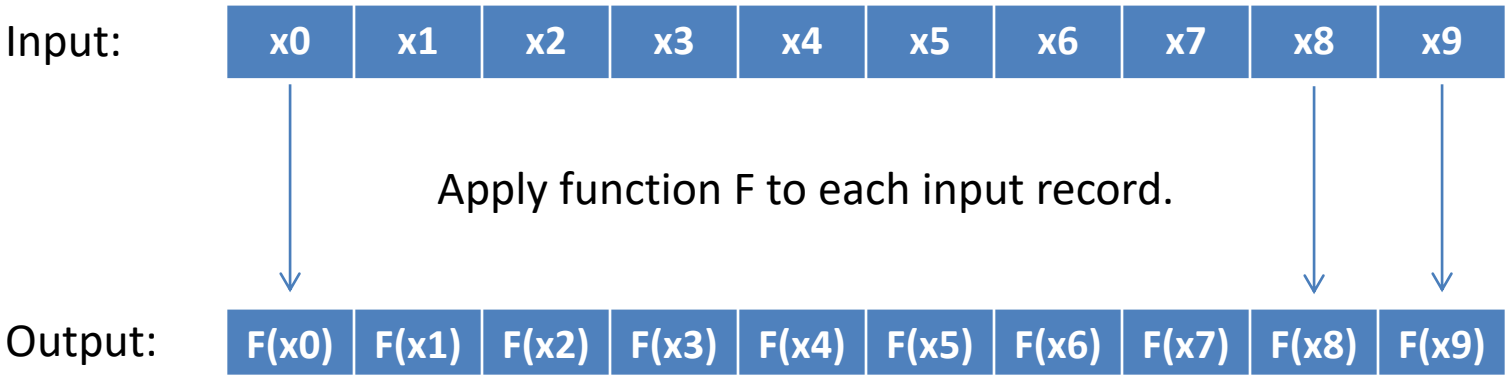
Do We Need Order Inversion in Spark?

- Challenge: try to solve the relative-frequency counting problem for the (species, color) data, using DataSet and pair RDD in Spark. Then check how the program is executed by Spark.
  - Is the execution more like the Pairs or more like the Stripes approach?
  - Is it like the order inversion implementation? If not, can you change that?
- Do we need order inversion in Spark?
  - Make sure you understand what aspects of the program order inversion improves, compared to Pairs and Stripes.
  - Then explore if the same improvements matter for Spark. If so, can they be achieved in Scala?

Next, we will explore a variety of useful "small" algorithms.

Per-Record Computation

- Per-record computation is easy to parallelize and fairly common:
  - The relational selection operator filters out records using a predicate. For example, from a dataset of flights, it can return all flights into Boston.
  - The relational projection operator removes fields from a record. For example, it could remove the flight arrival time field if it is not needed for the analysis.
  - The Unix grep command finds all lines in a (text) file that match a user-defined search pattern such as "Northeastern."
- **Task**: Given function F(), transform each input record x to F(x).
- **Solution**: Each task applies F() to its input records one-by-one. No shuffling is needed. For operators that filter records, e.g., selection and grep, F(x) conceptually returns NULL if x is filtered out.



Implementations

DBMS (SQL): Assume the input relation R has schema (A1, A2, A3)

Generic solution:	SELECT F(A1, A2, A3) FROM R
Selection:	SELECT * FROM R WHERE F(A1, A2, A3)
Projection (on A1):	SELECT A1 FROM R

MapReduce:

```
// Generic and projection
map( …, x )
  emit( F(x) )

// Selection
map( …, x )
  if F(x) then emit( x )
```

Spark:

```
myRDD.map(x => F(x))        // generic
myRDD.filter( F(x) )        // selection

myDS.map(x => F(x))         // generic
myDS.filter( F(x) )         // selection
myDS.select("A1")           // projection
```

Discussion

- The MapReduce implementation is a Map-only job. To specify this, the number of Reducers must be set to zero.
  - Look at the grep program from the Miner/Shook book on MapReduce design patterns: http://khoury.northeastern.edu/home/mirek/code/DistributedGrep.java
- The MapReduce, Spark, and generic SQL solution read the entire input. If every input record must be processed anyway, then this is the best one can do.
- For the selection operator, we are only interested in the matching records. Database systems provide index structures that can significantly reduce access cost for conditions that select only a small fraction of the input. MapReduce and Spark do not have such indexes. In a future module we will discuss HBase, a distributed key-value store that supports index-like lookups in a distributed system.

Content-Based Data Splitting

- Consider product reviews by users of an online shopping site, stored as records with schema (userID, isPreferred, productID, productCategory, review). An analyst might be interested in exploring reviews by preferred users separately from those by regular users. Similarly, a product-centric exploration might require training of separate data mining models for different product categories.
- **Task**: Partition the input into p separate subsets based on properties of the input records.
- **Solution**: Like per-record computation, each task goes through its input records one-by-one, sending the record to the desired output channel. No data shuffling is needed.
- This approach can also be used for problems where some input records are assigned to zero or more than one partitions.

Implementation in MapReduce

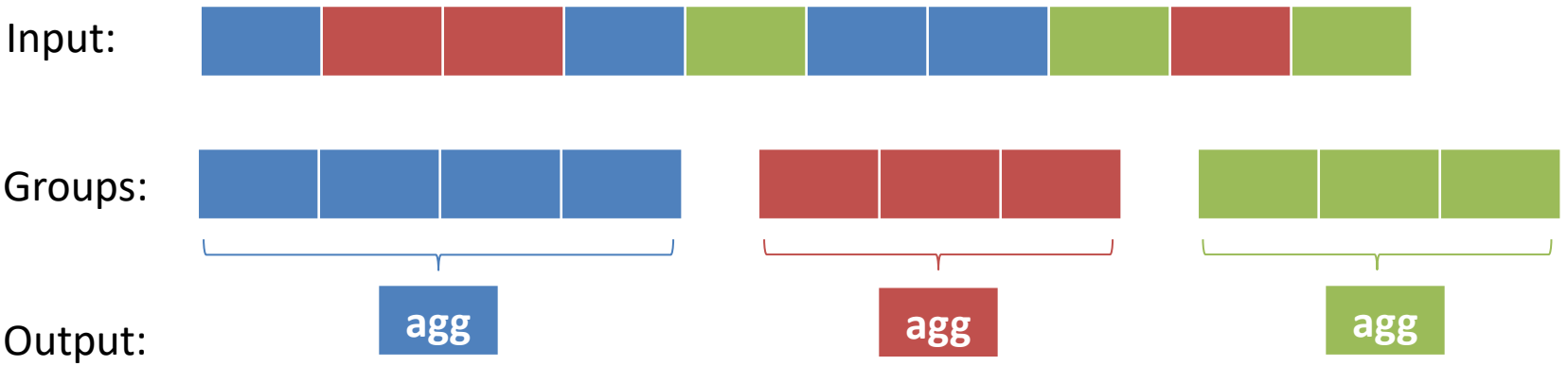
- A simple Map-only job can solve this task. It uses the MultipleOutputs class to write to different output files. Each of them can store records of a different type, e.g., one might store text data, another pairs of integer numbers. The Map function determines the partition the input record belongs to, then emits it to the appropriate output file using MultipleOutputs.write().
- For m Map tasks and p output partitions, this program will generate m·p output files. (Each Map task writes to its own set of p output files!) If these files are too small, they can be concatenated using HDFS file-system commands.
- Check out the example code from the Miner/Shook book for a program that parses XML documents to assign each to one of four partitions based on the "Tags" element: http://khoury.northeastern.edu/home/mirek/code/Binning.java

Discussion

- The elegance of the MapReduce program lies in reading the input only once and generating all partitions concurrently.
- In SQL or Spark, there is no equivalent construction. Instead, one could specify each partition with its own query.
  - The DBMS and Spark optimizer should automatically determine that all partitions can be generated with a single pass through the input data.

Grouping and Aggregation

- Data can be partitioned without shuffling only if the possible partitions are known in advance. For aggregation and when the partitions are not known, shuffling is needed. We have seen this for Word Count. More examples:
  - An inverted index for a document collection returns the identifiers of all documents that contain a given search string. To create this index for the Internet, for each word w, generate the list of URLs where w occurs.
  - For a graph, generate the inverted graph in adjacency-list format.
- **Task**: Partition the input on a key and compute an aggregate of the values in each partition.
- **Solution**: The first round filters records and removes irrelevant attributes. Then the shuffle phase ensures that all records with the same key end up together, so that the next stage can perform the per-group aggregation.



Implementations

DBMS (SQL): Assume the input relation R has schema (Key, Val)

SELECT Key, myAGG(Val) FROM R GROUP BY Key

MapReduce:

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

reduce( key, [val1, val2,…] )
  compute agg = myAGG(val1, val2,…)
  emit( key, agg)
```

Spark:

myPairRDD.aggregateByKey(aggFunction)

myDS.groupBy("Key").agg(aggFunction)

Grouping RDDs in Spark

- groupByKey transformation: for a pair RDD of type (K, V), it returns an RDD of type (K, Iterable[V])—one group per key.
  - All values for a key are collected in memory. When computing aggregates, it usually is more memory-efficient to use aggregateByKey, reduceByKey, or foldByKey instead.
- group(keyGenFct) transformation groups an ordinary RDD by extracting a key for each element and then grouping by key.
  - For an RDD of type T, keyGenFct: T => K generates keys of type K. Grouping is applied to those keys.
  - rdd.groupBy(f) is equivalent to
    rdd.map(x => (f(x), x)).groupByKey().

Grouping and Aggregation in Spark

- combineByKey transformation is a powerful function to group and aggregate elements of a pair RDD of type (K, V). It has several parameters:
  - createCombiner: V => C creates the first combined value of type C from the first key's value in each partition. It is used to set the initial value, e.g., v -> (v,1) for AVG.
  - mergeValue: (C, V) => C merges the other values for the same key *in the same partition* into the combined value, e.g., ((s,c), v) -> ((s+v), (c+1)) for AVG.
  - mergeCombiners: (C, C) => C merges combined values across partitions, e.g., ((s1,c1), (s2,c2)) -> (s1+s2, c1+c2) for AVG.
  - partitioner is required and determines the resulting RDD's partitioning. If it is the same as the current Partitioner, then no shuffling is performed. In that case, mergeCombiners would not be executed.
  - mapSideCombine: Boolean = true specifies whether to merge combined values in partitions before shuffling.
  - serializer: Serializer = null allows use of a custom serializer, instead of the default one specified in Spark configuration parameter spark.serializer.
- It generalizes and is used to implement aggregateByKey, groupByKey, foldByKey, and reduceByKey.

Discussion

- For distributive and algebraic aggregates one can use a Combiner or in-Mapper combining in MapReduce and in Spark, the corresponding functions that aggregate before shuffling. Examples for such aggregates are sum, count, average, minimum, maximum, and standard deviation. Combining is not applicable to holistic aggregates, e.g., median, except in the form of simple compression. For instance, one can replace set {A, A, B, A, B} by the more compact {(A,3), (B,2)}.
- Consider using a custom Partitioner for load balancing purposes.
- Consider secondary sort to simplify the Reduce computation, e.g., to guarantee an increasing order of values in the Reduce input list for finding the minimum.

Global Aggregation: Standard Approach

- Sometimes one would like to compute a single "global" aggregate for the entire input, e.g., the average delay over all flights. This can be done like grouping-and-aggregation.
  - In MapReduce, each Mapper computes the per-split aggregate, using in-Mapper combining or a Combiner. Then these values are associated with a "dummy" key and aggregated by a single Reduce call.

DBMS (SQL): Assume the input relation R has attribute A

```
SELECT myAGG( A ) FROM R
```

MapReduce:

```
map( key k, val v )
  emit( dummy, v )

reduce( dummy, [val1, val2,…] )
  emit( myAGG(val1, val2,…) )
```

Spark:

```
myRDD.aggregate( …, aggFunction,… )

myDS.agg( aggFunction )
```

Global Aggregation with Global Counters/Accumulators

- The standard approach we just discussed requires shuffling. Can we avoid this, i.e., can we run the equivalent of a Map-only job? Based on what you have seen so far, this seems impossible, because a task aggregating a data partition cannot compute the global aggregate.
- Simple aggregates like sum and count can be computed with a Map-only job by exploiting Hadoop's global counter feature. It is available through the Counter class. Global counter variables can be defined in MapReduce user code and any task can increment them or set their value. No matter which task updates a counter, in the end it will reflect the updates performed by all completed tasks. In Spark, the corresponding feature are Accumulators.

Global Aggregation with Global Counters/Accumulators (Cont.)

- By using multiple global counters, one could also compute aggregates for multiple groups of input records. For this to work, the groups must be known in advance. For instance, assuming all airlines are known in advance, one can define counters for each of them. An input record would result in counter updates for the corresponding airline only.
  - Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/CountNumUsersByStateDriver.java It counts the number of users per US state using global counters.

Duplicate Removal

- Duplicate removal eliminates repeat occurrences of records in an input file. It is a special case of grouping-and-aggregation: identical records form groups and from each group exactly one representative is output.
- **Task**: Eliminate all duplicates from the input.
- **Solution**: Group by the record content, then emit a single representative for each group.
  - The equivalent of in-Mapper combining can be applied to remove duplicates within the same split before shuffling.
  - An appropriately defined comparator determines when two records are considered identical. This could be based on a subset of the fields of the records.

Implementation

DBMS (SQL):

```
SELECT DISTINCT * FROM R
SELECT DISTINCT column1, column2,… FROM R
```

MapReduce:

```
map( key k, val v )
  emit( (k, v), NULL )

reduce( (k, v), [NULL, NULL,…] )
  emit( k, v )
```

Spark:

```
myRDD.distinct()

myDS.dropDuplicates()
myDS.dropDuplicates( column1, column2,…)
```

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/DistinctUserDriver.java
- The program finds all distinct user IDs. Since the input records are in XML format, they are parsed to access the user ID element.

Random Sampling

- Random sampling is crucially important for big-data analysis. Working with a random sample lowers computational cost while often still producing a good approximation of the desired result. Small random samples are also useful for testing and debugging.
- **Task**: Sample a fraction of approximately p, $0.0 < p \leq 1.0$, of the input records *uniformly at random*.
  - This means each input record should have the same probability p of being selected for the output.
- **Solution**: Each data partition can be sampled independently with sampling rate p. A pseudorandom-number generator determines if the input record will be emitted: If the generator produces a floating-point number rnd in the range $0.0 \leq rnd < 1.0$, then the input record is emitted if and only if $rnd < p$. No shuffling is needed.
  - For an input set of n records, this approach does not guarantee to produce exactly $p \cdot n$ output records. However, when dealing with large numbers of records, the resulting sample size in practice will be very close.

Random-Number-Generator Subtleties

- Libraries for generating “random” numbers usually provide only pseudorandom-number generators. This means that the generator will be initialized by some seed, e.g., based on the system clock. Then it produces a deterministic sequence of values that “appear random.” For the same seed, the generator will produce the exact same sequence of numbers. Seed choices are not truly random by nature. Hence in practice one should ideally only rely on a single pseudorandom-number-generator instance. Whenever a random number is needed, it should be produced by that same generator instance. This will result in better randomness than instantiating many different generators.
  - If you use Java’s Math.random() method whenever you need a random number, your program will automatically follow the preferred approach: The first call of Math.random() creates a single new pseudorandom-number generator object, which is then used for all future calls to this method and is used nowhere else.
  - Since different tasks cannot share the same generator object, a distributed program will have a separate generator instance per task.
- When dealing with big data, the period of a pseudorandom-number generator matters. It is the longest sequence of numbers generated, before the sequence starts repeating. The period should be larger than the number of times the generator is called.

Implementation

DBMS (SQL):

```
SELECT * FROM R WHERE random() <= p
```

MapReduce:

```
map( key k, val v )
  // Assume random() produces a pseudorandom
  // number n in the range 0.0 ≤ n < 1.0
  if random() < p then emit( k, v )
```

Spark:

```
// Sample without replacement: set first
// argument to FALSE
myRDD.sample( FALSE, p )

myDS.sample( p )
myDS.sample( FALSE, p )
```

More Info on Sampling in Spark

- There are different sampling functions:
  - Transformation sample(withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong)
    - Fraction is the sampling rate, not the exact sample size.
  - Action takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong)
    - Returns exactly num elements
  - Action take(num: Int)
    - Scans RDD partitions until sufficiently many elements are returned. This does not randomly select elements from the entire RDD!

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/SimpleRandomSampling.java
- Notice how this program uses a single pseudorandom-number generator per Map task.

Random Shuffling

- Random shuffling, i.e., arranging records in a file in a random order, can be useful in many situations. After a file is randomly shuffled, each block will contain a random sample of records. Hence one can obtain a random sample of exactly k records by simply reading the first k records from the shuffled file.
- **Task**: Randomly shuffle a given input file. Each input record should have the same probability of ending up in any position in the shuffled file.
- **Solution**: We sort the input by a (pseudo-) random key, which is assigned by tasks in the first round of computation.
  - Instead of sorting, a simpler “random grouping” by the pseudorandom key would also suffice. For random keys, hash partitioning will create such a random grouping. Since each input record can still end up in any position with the same probability, there is no need for the more expensive (due to the quantile sampling step) and potentially less load-balanced (if quantiles are poorly approximated) range partitioning.

Implementation

DBMS (SQL):
SELECT * FROM R ORDER BY random()

MapReduce:
```
map( key k, val v )
  // To avoid duplicate keys, chose the random
  // number from a large domain, e.g., floating
  // point numbers between 0.0 and 1.0.
  emit( random(), (k, v) )

// If none of the random keys are identical, then
// the input list contains only a single record
reduce( rnd, [(k1, v1), (k2, v2),…] )
  for each (k, v) in input list
    emit( k, v )
```

Spark:

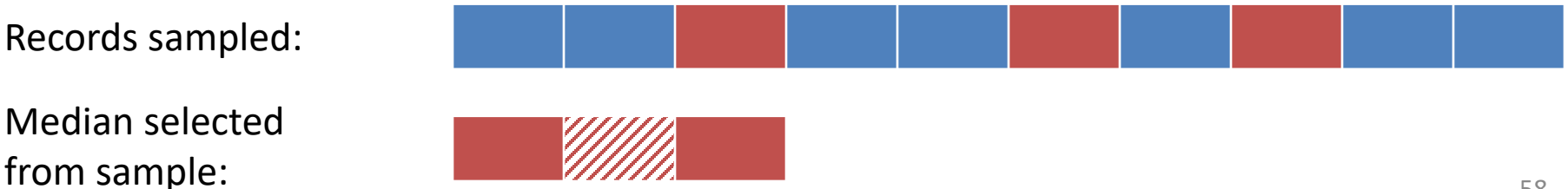
```
myRDD.map( x => (rand(), x) )
        .sortByKey()

myRDD.map( x => (rand(), x) )
        .groupByKey

myDS.sort( rand() )
```

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/AnonymizeDriver.java
- This program assigns a random integer as the key.

Approximate Quantiles

- Quantiles such as the median provide important information about a data distribution. For instance, the median housing price is a measure of wealth for a neighborhoods. Furthermore, range-partitioning based on quantiles results in balanced load distribution because the number of records between consecutive quantiles is the same (unless there are many duplicates).
- As discussed in a previous module, exact quantiles can be found by sorting the data and then picking the records at the corresponding positions, e.g., the record in the middle for the median. In practice, approximate quantiles often suffice. The approach discussed here can find approximate quantiles in a single pass over the input data.
- **Task**: Find approximate quantiles or percentiles.
- **Solution**: The main idea is to randomly sample input records in the first round. Then all sampled records are sorted by a single task in round 2. The approximate quantiles are collected from the sorted sample.
  - A good choice is to create a sample that is large, but still fits into the memory of the round-2 task.



Challenge Question

- What are the drawbacks of a sample that is too small or too large?

Challenge Question: Answer

- What are the drawbacks of a sample that is too small or too large?
  - The smaller the sample, the less data is available for determining the quantiles. This leads to poorer approximation quality.
  - On the other hand, if sample size exceeds the amount of memory, sorting it would be significantly more expensive.

Implementation in MapReduce

- The algorithm for approximate quantiles is shown below.
- Would secondary sort be helpful here? It eliminates the need for sorting in user code. And if the number of records in the input list is known, reduce() can scan through the sorted list and pick the quantiles from the appropriate positions on-the-fly.
  - Then the reduce function would not need more than a few bytes of memory to hold the currently read input record and to maintain a counter for the position in the list.
  - Unfortunately, it is not clear how to obtain the size of the input list without reading it twice. (This is not supported by the iterator, therefore, to read it twice, the list must be copied into memory.) A global counter could track the number of sampled records, but it cannot be read by the Reduce task of the same job. Order inversion could help here.

```
map( key k, val v )
  // Assume random() produces a
  // pseudorandom number n in the
  // range 0.0 ≤ n < 1.0
  if random() < p then emit( dummy, (k, v) )
```

```
reduce( dummy, [(k1, v1), (k2, v2),…] ) {
  copy all records from the input list into array A
  sort array A

  for each quantile position i in sorted array A
    emit( A[i] )
}
```

Approximate Quantiles using Order Inversion

- The job has one Reduce task. The key is (k1, k2) where k1 is “sample” or “count” and k2 is a sampled record (k,v) or a count value, respectively. The key comparator first compares k1, using “count” < “sample”; if equal, it compares k2, where count values are ordered arbitrarily and (k,v) tuples are ordered based on their given ordering property. A grouping comparator ignores the k2 component of the key.

```
map( key k, val v ) {
  if random() < p then {
    // Emit the sampled record as before
    emit( “sample”, (k, v) )

    // This is needed to get the sample size
    emit( “count”, 1 )
  }
}
```

```
Class Reducer {
  sampleSize = 0

  reduce( flag, [x1, x2,…] ) {
    if flag = “count” then
      for all x in input list
        sampleSize += x
    else    // flag = “sample”
      position = 0
      for each x = (k,v) in input list
        position++
        if position/sampleSize is a quantile position
          emit( k, v )
  }
}
```

Implementation in Spark

- We could implement the MapReduce algorithm equivalently in Spark Scala. It first calls the sample function, then collects the sampled data at the driver. The driver then performs sorting and quantile selection on the corresponding collection.
  - Alternatively, after sampling, we can map each record x to (dummy, x), then use repartitionAndSortWithinPartitions to sort the single partition for the dummy key.
- Spark DataFrames also support the approxQuantile function. It uses a one-pass algorithm with approximation-quality guarantees. See [M. Greenwald and S. Khanna. Space-efficient Online Computation of Quantile Summaries. In Proc. ACM SIGMOD Int. Conf. on Managament of Data, pages 58-66, 2001] for details.

Implementation in a DBMS

- The SQL standard does not seem to include a function for computation of approximate quantiles.
- Database vendors offer their own versions of quantile and percentile computation functions. For example, Vertica (as of September 2018) supports an approximate algorithm based on a research paper [T. Dunning and O. Ertl. Computing Extremely Accurate Quantiles Using t-Digests. 2017]

Top-K Records

- In addition to quantiles, one can gain valuable information about a big dataset from the K most important records. This could be the top-K largest (or smallest) records based on some field or attribute, e.g., the most active users, people who spend the most money or time, or users with the most friendship links.
- **Task**: Find the K records that have the largest value of some attribute of interest.
- **Solutions**:
  - For large K, sort the input and then select the first K records.
  - For small K, sorting can be avoided. The main idea is to scan the input only once. In the first round, each task finds the local top-K in its partition. All local top-K are sent to a single task that merges them into the final result. Correctness is guaranteed, because if a record is in the global top-K, it must be in one of the local top-K.

MapReduce Implementation

- The algorithm below could also use the secondary sort design pattern to simplify the reduce function. With secondary sort, the reduce function simply picks up the first K records in the input list.

```
Class Mapper {
  localTopK

  setup() { init localTopK }

  map( v )
    if (v ranks above the last value in localTopK)
      // Adding v must also evict the now
      // (K+1)-st value from localTopK
      localTopK.add( v )

  cleanup()
    for each v in localTopK emit( dummy, v )
}
```

```
reduce( dummy, [v1, v2,…] )
  init globalTopK

  for each value v in input list
    if (v ranks above the last value in globalTopK)
      // Adding v must also evict the now
      // (K+1)-st record from globalTopK
      globalTopK.add( v )

  for each value v in globalTopK
    emit( v )
}
```

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/TopTenDriver.java
- This program finds the top-10 users with the greatest reputation. Since the input records are in XML format, they are parsed to access the reputation element.

Implementation in Spark and DBMS

DBMS (SQL): Assume the input relation R has attribute A

```
SELECT * FROM R
ORDER BY A
FETCH FIRST K ROWS ONLY
```

Spark:

```
// The RDD needs to have an implicit Ordering for the type of objects stored
myRDD.top( K )
myRDD.takeOrdered( K )

myDS.sort( A ).head( K )
```

More Info about Top-K in Spark

- takeOrdered(k) and top(k) actions return the k first and last elements of the RDD, respectively. Ordering is determined by an implicit Ordering object defined in scope.
- When applied to pair RDDs of type (K, V), order would not be based on key, but on (K, V) tuples. To change this, one needs to have an implicitly defined Ordering[(K, V)] object in scope. This is the case for simple key and value types by default.
- Implementation in Spark corresponds to the top-K MapReduce algorithm: it takes the top-K elements from each partition, then merges them into a single top-K list.
  - Note that the final result will end up in the driver's memory.

Summary

- Big data analysis often involves partitioning, sampling, and summarizing of data. Parallel solutions in MapReduce and Spark are comparably straightforward, sometimes not requiring any data shuffling.
- In most cases, design patterns introduced previously, e.g., in-Mapper combining and secondary sort, can significantly improve performance or reduce coding effort.

References

- M. Greenwald and S. Khanna. Space-efficient Online Computation of Quantile Summaries. In Proc. ACM SIGMOD Int. Conf. on Managament of Data, pages 58-66, 2001
  - https://scholar.google.com/scholar?cluster=6184540005789557130&hl=en&as_sdt=0,22