

Common Algorithm Building Blocks

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

- 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.
 - For example, we can distribute count statistics such as the number of records in a dataset, e.g., to enable relative frequency computation (fraction of records of some type in the data).
- The distributed `file cache` in MapReduce allows broadcasting of larger read-only state. In Spark, this is supported by `broadcast variables`.
 - We will see an example when discussing the Replicated Join algorithm.
- 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 works correctly independent of the order in which the updates are applied. This greatly simplifies distributed consensus.

Other Options for Global State?

- How about larger objects managed by the application master, e.g., a central priority queue for algorithms such as single-source shortest path (discussed in a future module)?
 - 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.
 - However, 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?
- Yes! 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 we will explain 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. Given a collection of such records, our goal is for each species and color to estimate the conditional probability of the color, given the species.
 - For example, mathematically the probability of the Northern Cardinal (N.C.) being red is defined as $P(\text{color} = \text{red} \mid \text{species} = \text{N.C.})$.
- We can estimate such probabilities from big data by counting the appropriate quantities. To estimate $P(\text{color} = C \mid \text{species} = S)$, we need the following:
 - Frequency count $f(S)$, i.e., the number of records that are observations for species S . In statistical terms, this is called a **marginal**.
 - Frequency count $f(S, C)$, i.e., the number of records that are observations matching both species S and color C . In statistical terms, this is called a **joint event**.
 - In the example, we estimate $P(\text{color} = \text{red} \mid \text{species} = \text{N.C.})$ by dividing the number of red Northern Cardinal observations by the total number of Northern Cardinal observations (including all colors!), i.e., as $f(\text{N.C., red}) / f(\text{N.C.})$
- This analysis is an example for estimating **relative frequencies**, a common data mining task. Another example 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 simply emits the record with the species S as the key, and the 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 the 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 the memory size? This would not happen for the colors here, but it might for other problems with more columns.
 - The granularity of the Reducer workload is limited by the number of different species. What if we have more machines than species? How can we keep all of them busy?

Colors

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 how smaller units of work could become problematic, let us consider a program that uses both species and color as the intermediate key.
- This program is virtually identical to Word Count. For input record (species S , color C), Map emits $((S, C), 1)$.
 - Combiners and in-Mapper combining can be applied here as well.
- For key (S, C) , Reduce computes $f(S, C)$, the frequency count of that species-color combination. Like Word Count, virtually no memory is used and the fine granularity enables up to $\#species \cdot \#colors$ different Reduce tasks.
- So, does this approach have any drawbacks?
 - Yes! It is not clear how to compute the marginal $f(S)$. A Reduce call that computes $f(S)$ would have to access $f(S, color)$ 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 set 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 for some species S the 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 have to make sure that *all keys $(S, *)$ 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 ?
 - 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?
 - 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 have to 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 limitation of all attempts so far has been that they 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?

```
map( ..., observation: (species S, color C) )  
  emit( (S, C), 1 )
```

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

```
  for all n in input list do  
    frequency += n
```

```
  // Here we assume that  $f(S)$  is “magically”  
  // known. How can it be made available  
  // without requiring a separate MapReduce  
  // job for pre-computing it?  
  emit( (S, C), frequency /  $f(S)$  )
```


Solution Using Order Inversion

- Consider the program below. Like the Pairs-based approach, it uses both species and color for the intermediate key. In addition to $((S, C), 1)$, Map also emits another record $((S, \text{dummy}), 1)$, which will be used to separately compute $f(S)$. By decoupling these computations, MapReduce's property of executing Reduce calls in key order can be exploited to control the order of frequency computations. In particular, by defining a key comparator that sorts color "dummy" before each real color C , it is guaranteed that $\text{reduce}((S, \text{dummy}), \dots)$ would be executed before $\text{reduce}((S, C), \dots)$ for any other color C in the same Reduce task.
- A custom Partitioner that partitions only based on the species, guarantees that all joint events for a species-color combination are processed in the same task as the marginal for that species.
- 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 following Reduce calls for $f(S, C1)$, $f(S, C2)$,... Since the key comparator ensures sorting by species first, it is also guaranteed that the right marginal $f(S)$ is available even if multiple species are assigned to the same Reduce task.
- Notice that the Reduce task now has a small constant-size (i.e., independent of the number of species and colors) memory footprint. It only 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
- 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 )  
    }  
  }  
}
```

Discussion

- How does this new solution compare to the previous attempts?
 - Good: It reads the input data only once, like the Stripes-based approach.
 - Good: It requires virtually no memory, like the initial Pairs-based approach.
 - Potentially bad: Map duplicates every input record, therefore in the worst case two copies of the big input data set are transferred from Mappers to Reducers. In practice, use of a Combiner or in-Mapper combining can often dramatically reduce this data transfer.
 - Bad: Like the Stripes-based approach and the failed fixes of the single-job Pairs-based approaches, Reduce granularity is potentially very coarse, because it is determined only by the species.
- Interestingly, this approach in principle supports a finer Reduce task granularity at the cost of greater 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 $(S, C1)$, $(S, C2)$, $(S, C3)$,... can be distributed over k different Reduce tasks. The Partitioner has to 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, this k -fold data duplication can result in poor performance unless combining is very effective.

Multiple Vertical Partitions (on Color)

```
map( ..., observation: (species S, color C) ) {  
  emit( (S, dummy1), 1 )  
  emit( (S, dummy2), 1 )  
  ...  
  emit( (S, dummyk), 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 have to rely on either (1) larger and more complex data structures to bring the right data together (e.g., the hashmap structure H for a Stripe) or (2) more MapReduce jobs to compute the intermediate results in the appropriate phase.
- In some sense, order inversion turns a synchronization problem into an ordering problem: MapReduce has no explicit synchronization primitives. Hence order inversion relies on the key sort order to enforce computation order. A custom Partitioner assigns the appropriate data to each Reduce task and the Reducer maintains small task-level state across Reduce invocations.
- Advantages: This approach enables the use of simpler data structures and less Reducer memory, without requiring additional MapReduce phases. It also enables controlling the granularity of Reduce tasks at the cost of greater data replication.
- Disadvantages: Due to data replication in the Mappers, performance can suffer unless an effective Combiner or in-Mapper combining significantly reduces data transfer from Mappers to Reducers.

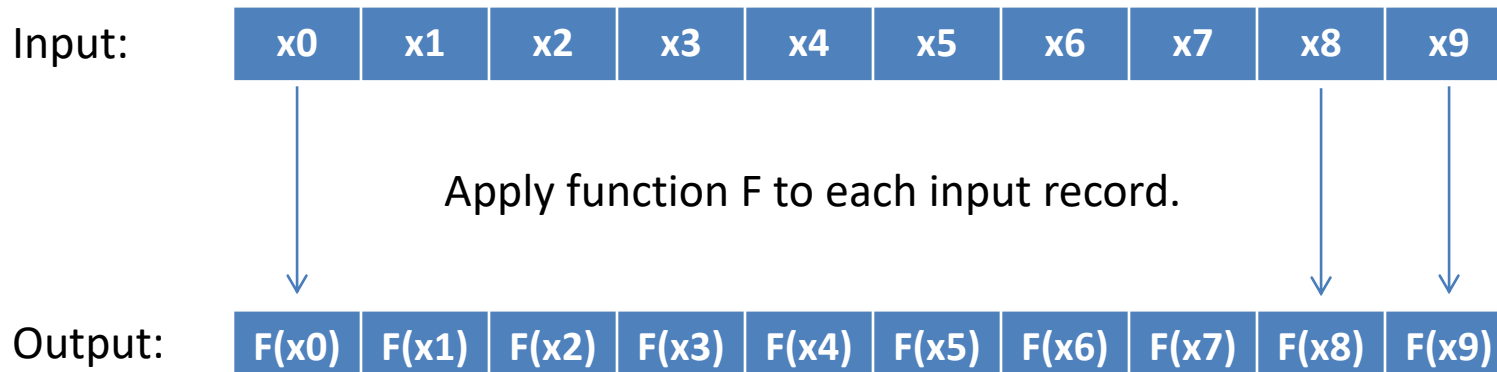
Do We Need Order Inversion in Spark?

- 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 a trivial problem for parallel computation, but very useful in many applications:
 - The relational **selection** operator filters out records based on a user-provided predicate. For example, from a data set of flights, it can select the flights originating from Boston.
 - The relational **projection** operator removes fields from a record. For example, from a flight data set it could remove the arrival time field if that is not needed for the analysis task.
 - The Unix **grep** command finds all lines in a (text) file that match a user-defined string search pattern such as “Northeastern.”
- **Task:** Transform each input record independently by applying some function $F()$ to it.
- **Solution:** Each task applies function $F()$ to its input records one-by-one, emitting the results. No data shuffling is needed. For operators that filter out records, e.g., the selection operator and grep, $F(x)$ conceptually returns NULL for input records x that are 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  
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 has to be set to zero.
 - Take a look at the grep program from the Miner/Shook book on MapReduce design patterns:
<http://www.ccs.neu.edu/home/mirek/code/DistributedGrep.java>
- Note that the generic SQL solution and the MapReduce and Spark programs read the entire input data set. If every input record has to be processed anyway, then this is the best one can do. However, for the selection operator, we are only interested in the matching records. Database systems provide **index structures** that can significantly reduce access cost for highly selective conditions, i.e., conditions that only select a very 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 storage system that can support index-like lookups in a distributed system.

Content-Based Data Splitting

- Consider data about 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 the different product categories.
- **Task:** Partition the input dataset into p separate sub-sets based on properties of the input records.
- **Solution:** Similar to 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 of the p partitions.

Input:



Output:



Implementation in MapReduce

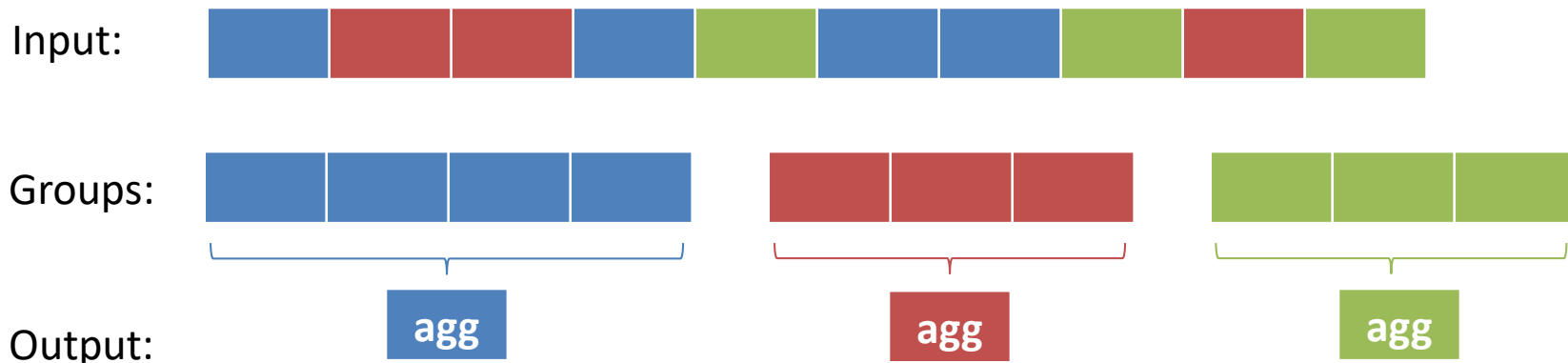
- A simple Map-only job can solve this task. It uses the `MultipleOutputs` class to write to p different output files. Note that each of these output files can store records of a different type, e.g., one might store text data, another pairs of integer numbers. The Map function determines the partition the input record belongs to, then emits it to the appropriate output file using `MultipleOutputs.write()`.
- Note that for m Map tasks and p desired output partitions, this program will generate $m \cdot p$ output files. (Each Map task writes to its own set of p output files!) If these files are too small, they can be concatenated easily 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 different partitions based on the “Tags” element:
<http://www.ccs.neu.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 such an index for the World Wide Web, for each word w , generate the list of URLs where w occurs.
 - For a graph given as a set of edges, generate the inverted graph stored as an adjacency list.
- **Task:** Partition the input data set on a key and compute an aggregate of the values in each partition.
- **Solution:** The first round of computation can filter records or remove 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 )
```

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 the use of 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 dataset, e.g., the average delay over all flights. This can be done similar to the grouping-and-aggregation approach.
 - 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 single file split cannot compute the global aggregate.
- Simple aggregates, in particular 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 have to be known in advance. For instance, assuming all airlines are known in advance, one can define the sum and count 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://www.ccs.neu.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://www.ccs.neu.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's importance for big data analysis cannot be overemphasized. When data is too big to handle with available computational resources, working with a random sample can reduce 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 split 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 $0.0 \leq \text{rnd} < 1.0$, then the input record is emitted if and only if $\text{rnd} < p$. 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.

Input: 

Output:  Sampling rate $p = 0.3$

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 “look 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 to produce a new pseudorandom number.

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  $0.0 \leq n < 1.0$   
  rnd = random()  
  if rnd < 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 )
```


Real Code

- Check out the example code from the Miner/Shook book at <http://www.ccs.neu.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) TotalOrderPartitioner, i.e., 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.
```

```
rnd = random()  
emit( rnd, (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://www.ccs.neu.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 good indicator for distinguishing rich and poor neighborhoods. Furthermore, range-partitioning based on quantiles will result in balanced load distribution because by definition the number of records between consecutive quantiles is about the same.
- 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.

Records sampled:



Median selected
from sample:



Challenge Questions

- 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. Use of [secondary sort](#) would eliminate the need for sorting in user code.
- In theory, with secondary sort, the Reduce function does not even need to load all records from the input list into memory. If the total number of records in the input list is known, it can simply scan through the (already sorted!) list and pick the corresponding quantiles from the appropriate positions on-the-fly.
 - In this case, 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 has to be copied into memory.) A global counter could track the number of sampled records, but it cannot be read out by the Reduce task of the same job.

```
map( key k, val v )  
  // Assume random() produces a  
  // pseudorandom number n in the  
  // range  $0.0 \leq n < 1.0$   
  rnd = random()  
  if rnd < 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] )  
}
```

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 would perform the 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 Management 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. 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 data set by exploring the K most important records, based on some notion of importance. In particular, it is often insightful to look at the top-K largest (or smallest) records based on some field or attribute. Common examples include 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.
- **Solution:**
 - Sort the input data; then select the K largest records from the sorted file. This works well for very large K.
 - For smaller values of 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 file split. All these local top-K lists are then sent to a single task that merges them into the final result. It is guaranteed that this approach will find the exact global top-K, because if a record is in the global top-K, it has to be in the corresponding 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( ..., val v )  
    if (v ranks above the last value in localTopK)  
      // Adding v should 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 should evict the now  
      // (K+1)-st record from globalTopK  
      globalTopK.add( v )  
  
  for each value v in globalTopK  
    emit( v )  
}
```

Implementation in Spark and DBMS

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

```
SELECT * FROM R ORDER BY A LIMIT K
```

Spark:

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

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

Real Code

- Check out the example code from the Miner/Shook book at <http://www.ccs.neu.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.

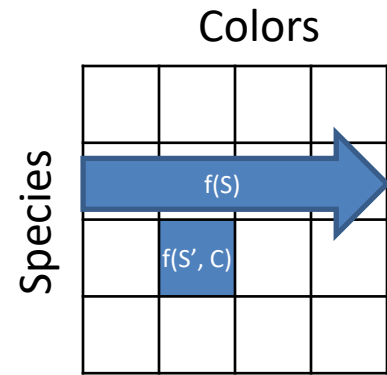
Summary

- 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 (in particular in-Mapper combining and secondary sort) can significantly improve performance or reduce coding effort.

Review Quiz: Question 1

- Which of the following problems can be solved without a shuffling phase?
 1. Sample randomly about 10% of the input records.
 2. In a file containing records of type (airlineID, delay), find all flights of US Airways.
 3. Remove all duplicates from a given input file.
 4. Given a file “Wrote” containing (authorID, bookID) records, output a file “WrittenBy” containing (bookID, authorID) records.

Question 2



- Recall the relative frequency count problem: for each combination of species S and color C , compute $f(S, C) / f(S)$. For each scenario below, choose the “better” solution between the Stripes and Order Inversion approach.
 - Very large $|S|$ (number of species) and small $|C|$ (number of colors).
 - Small $|S|$ and small $|C|$.
 - Very large $|S|$ and very large $|C|$.

Reminder: Stripes Program

```
// 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 )  
}
```

Reminder: Order Inversion Program

```
map( ..., observation: (species S, color C) ) {  
    emit( (S, dummy1), 1 )  
    emit( (S, dummy2), 1 )  
    ...  
    emit( (S, dummyk), 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 )  
        }  
    }  
}
```

References

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