1. Order Inversion Design Pattern

- In contrast to traditional parallel programming models, MapReduce does not provide explicit synchronization primitives to the programmer.
  - Examples for synchronization primitives are locks, barriers, and (synchronous) message sending and receiving. (These will be discussed more in a future module.)
- Explicitly managing synchronization in a distributed system is generally considered complex and challenging: Whenever multiple processes interact in modifying data concurrently, careful coordination is required to (1) guarantee correctness under all circumstances and (2) achieve good performance. Freeing the programmer from worrying about these low-level details is one of the strengths of MapReduce, and a reason for its popularity.
- However, in the hands of an experienced expert, explicit synchronization can be a powerful tool for creating efficient parallel programs.
- Let's see how we can perform some form of synchronization in MapReduce. Doing so requires a clever use of Map, Reduce, Partitioner, and key ordering.

1.1 Example

- The need for explicit synchronization will be illustrated using an example from statistical data analysis and data mining: Consider a crowd-sourcing protocol where citizen scientists report bird species they observe. For the sake of simplicity, assume participants report a (species, color) record every time they see a bird.
- Given a large collection of such records, our goal is for each species and color to estimate the probability of the color for that species. More formally, the goal is to estimate the conditional probability of a color, given a species.
  - For example, mathematically 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. In particular, to estimate P(color = C | 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(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 example problem with the same structure is to compute the normalized word co-occurrence vector for each word in a document collection. Intuitively it measures for each word, which other words occur frequently near it.

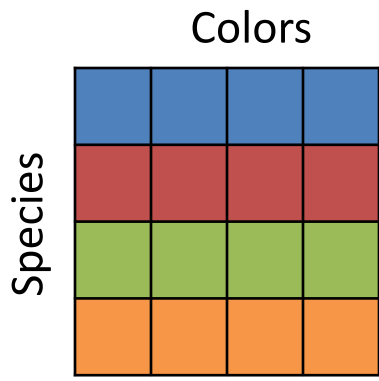
1.2 Obvious Solution Using "Stripes"

- Notice that both frequency counts, f(S) and f(S, C), count per species. Hence the species presents itself as an obvious choice for intermediate key.
- Starting with this observation about the key, the entire MapReduce program just "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 combining approach could then aggregate these counts.
- 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 can also keep track of the marginal f(S) for the species.

```
// Note: If no combining approach is used,
// Map could simply 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 )
}
```

1.2.1 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 cell in this table, indexed by the combination of a 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. (We will discuss this and other partitioning options of a multidimensional space in more detail in a future module.)
- 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. More precisely, all the data needed for computing f(S, C)/f(S) for species S is in the corresponding stripe. And the stripe contains no additional irrelevant data.
- So, is there a drawback to this approach? There are indeed two major limitations:
  - First, what if data structure H in Reduce exceeds the size of the available memory? This would not happen for the color example, but it might for other problems where the table has many more columns.
  - Second, 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?

1.3 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's 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 times #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 finer work partitioning of Pairs?

1.3.1 Fixing the Pairs-Based Approach, Attempt 1

- Let’s try to fix the Pairs-based approach so that is 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 partitioning function that assigns a key (S, C) to a Reducer solely based on the S field, ignoring the value of the C field of the key.
- While the custom Partitioner guarantees that all records for species S will be processed in the same educe 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 2: Does either of these approaches really give us a better solution than the Stripes approach?]

Challenge Question Answers

- Q1: Possible answer 1: The individual Reduce calls for keys (S, C1), (S, C2), (S, C3),… could be turned into a single Reduce call for species S by using a grouping comparator. (Review the secondary sort design pattern.)

- Q1: Possible answer 2: State can be maintained across the different Reduce calls for keys (S, C1), (S, C2), (S, C3),… 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.)

- Q2: No. Unfortunately 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 concurrently to 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 just “simulate” the Stripe-based approach.

1.3.2 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, then the Pair-based approach would be trivial as shown by the code below. Unfortunately, in reality f(S) is not magically known and needs to be computed.
- It turns out that this can be done without a separate MapReduce pre-processing step.

```
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.
  emit( (S, C), frequency / f(S) )
}
```

1.4 Solution Using Order Inversion

- Consider the MapReduce program below. Like the Pairs-based approach, it uses both the species and the color for the intermediate key. In addition to the usual ((S, C), 1), Map also emits another record ((S, dummy), 1). The former will be used to compute f(S, C), while the latter will be used to compute f(S). By de-coupling these computations, MapReduce's property of executing Reduce calls in key order can now be exploited to control the order in which the different frequencies are computed. In particular, by defining a key comparator that sorts color "dummy" before each real color C, it is guaranteed that reduce((S, dummy),…) would be executed before reduce((S, C),…) for any other color C in the same Reduce task.
- A custom partitioner that partitions only based on the species component, 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, called marginal in the algorithm ,to keep track of f(S). Since the Reduce call for the dummy color happens first, it is guaranteed that 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 (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.

Code for Order Inversion

```
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 )
    }
  }
}
```

1.4.1 Discussion

- How does this new solution compare to the previous attempts?
  - Good: It reads the big input data only once, like the Stripes-based approach.
  - Good: It requires virtually no memory, like the simple 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 "fixed" single-phase Pairs-based approaches, Reduce granularity is potentially very coarse, because it is determined only by the species.
- Interestingly, this approach supports a finer Reduce task granularity at the cost of greater data replication in the Map phase. The code below 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 simply has to ensure that each of these tasks receives one of the (S, dummyi) 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.

Code for K-fold Duplication Approach

```
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 )
    }
  }
}
```

1.5 Order Inversion Design Pattern Summary

- This design pattern for controlling the order in which intermediate results are computed is called "order inversion." [Challenge question: 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 phases to compute the intermediate results in the appropriate phase.
- Intuitively, order inversion turns a synchronization problem into an ordering problem. More precisely, 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.

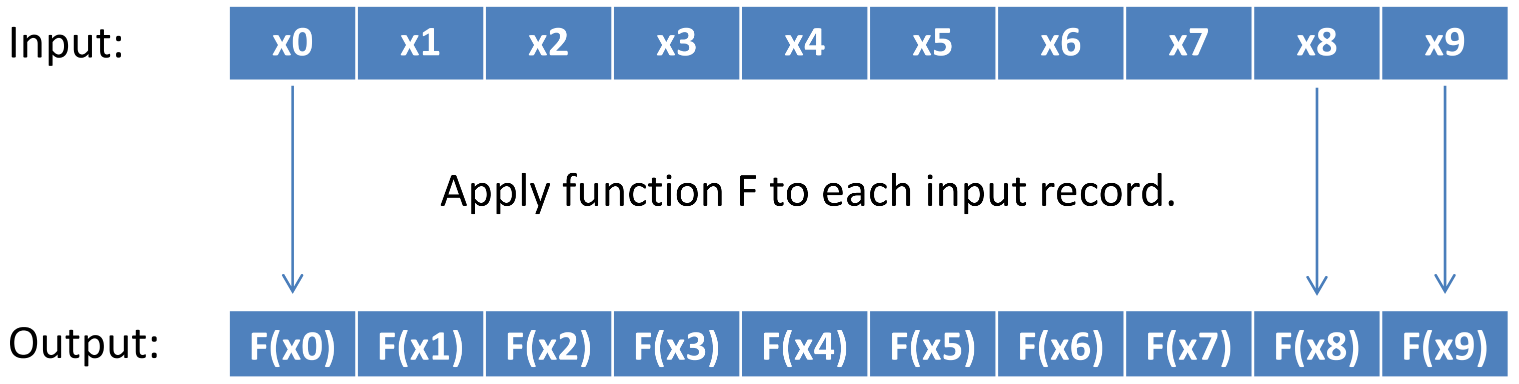
2. Utilities

- There are certain tasks that are almost always needed when analyzing data:
  - Sorting (discussed in a previous module)
  - Data partitioning
  - Grouping and aggregation
  - Sampling
- This section introduces efficient MapReduce programs for these tasks.

2.1 Per-Record Computation

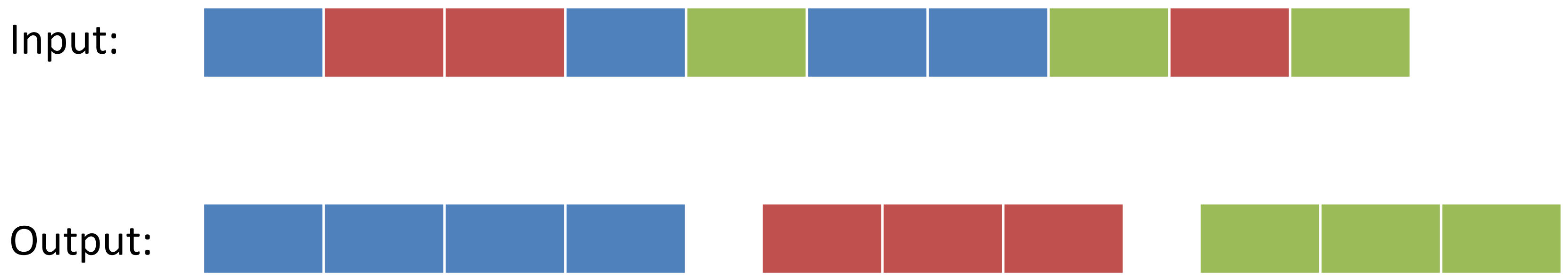
- Per-record computation is a trivial problem for MapReduce, but very useful in many applications. For example:
  - The relational selection operator filters out records based on a user-defined selection predicate. For example, from a data set of flights it can select the flights originating from Boston.
  - 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."
  - Invert all edges of a graph that is stored as a set of edges. More precisely, convert each edge (X, Y) into edge (Y, X). This is a common challenge in graph analysis.
- Task: Transform each input record independently by applying some function F() to it.
- Solution: A simple Map-only job can solve this task. The Map function performs the required computation on the input record and emits the result.
  - 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.
- Note that this program reads 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 does 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 the MapReduce ecosystem.

```
map( …, x )
  emit( NULL, F(x) )
```



2.2 Map-Only Data Partitioning

- 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. In both examples the given data set has to be partitioned into separate sub-sets.
- Task: Partition the input data set into p separate sub-sets based on properties of the input records.
- Solution: 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, the other 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 actually 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 easily using HDFS file system commands.
- This approach can also be used for problems where some input records are assigned to zero or more than one of the p partitions.



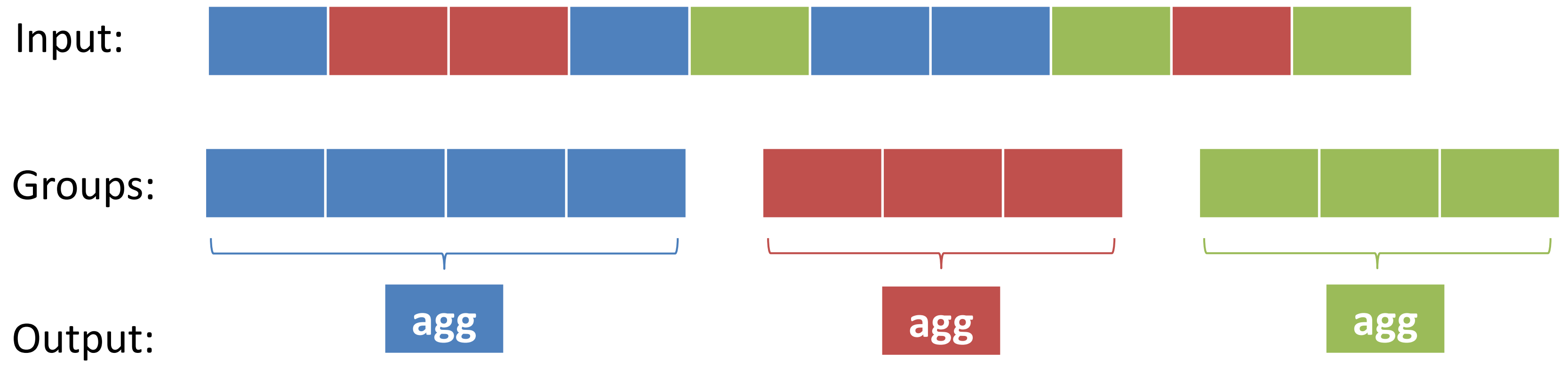
2.3 Grouping and Aggregation

- Data can be partitioned with a Map-only job only if the possible partitions are known in advance. When the partitions are not known or when one wants to compute an aggregate for each partition, Reduce is needed. This is a very common data analysis task, for example:
  - Word Count groups by the word string and computes the total count per word.
  - SQL's GROUP BY operator can be used to compute an aggregate such as the average delay for each airline. Flight data would be grouped by airline and then the average delay is computed for each group. GROUP BY does not need to know the airlines in advance, hence it would work correctly even if new airlines appear in the data.
  - An inverted index for a document collection returns the identifiers of all documents in the collection that contain a given search string. To create such an index for the World Wide Web, one has to do the following: for each word w on the Web, 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. Stated differently, for each node of the graph, generate the list of incoming edges.
  - Create a new object from the fields of multiple input objects. This can be useful when processing XML documents such that new documents are formed by combining tags from different input documents.
- Task: Partition the input data set and compute an aggregate for each partition.
- Solution: This task requires both Map and Reduce phase.
  - The Map function emits the input record, using the grouping attribute as the key and the record itself as the value. (Note that the grouping attribute could be removed from the value component as it is already contained in the key component.)
  - The Reduce function receives all records in the corresponding group and can then compute the desired aggregate function. Depending on the aggregate function, e.g., median, this computation might require multiple passes over the input list or a sufficiently large amount of heap space to load it into memory.
- As discussed before, for distributive and algebraic aggregates one can use a Combiner or in-mapper combining. 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. (Recall that a Combiner could 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.

2.3 Grouping and Aggregation

```
map( …, x )
  emit( x.groupingAttribute, x )
```

```
reduce( groupID, [x1, x2,…] ) {
  agg = compute some aggregate over the values in the input list
  emit( groupID, agg )
}
```



2.4 Global Aggregation

- Sometimes one would like to compute a single "global" aggregate for the entire input data set, e.g., the average flight delay over all flights. This can be done using the grouping-and-aggregation approach. [Challenge question: How exactly could that be done?]
  - Use of Combiner or in-mapper combining is essential for this approach, because all values are sent to a single Reducer.
- Can one find an even more efficient approach? In particular, can a global aggregate be computed with a Map-only job? Based on what you have seen so far, this seems impossible, because whenever there are multiple Map tasks, none of them can compute the desired result, because neither sees the entire input. Hence Reduce is needed to aggregate across output from different Map tasks.
- It turns out that 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.
- To compute the average flight delay with a Map-only job, define two counters: one to keep track of the sum of all flight delays, the other for the count. When processing a flight record, the Map function simply adds the delay to the first counter (for the sum) and value 1 to the second counter (for the count). The driver program can then read out the final counter values and return the desired average flight delay.
- By using multiple global counters, one could also compute aggregates for multiple groups of input records. However, 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. (If the groups are not known in advance, one has to fall back to the general grouping-and-aggregation program with Reduce phase.)
- Note that due to their global nature, these counters are maintained by the JobTracker. Hence tasks that update a counter have to inform the JobTracker. This centralized maintenance of global counters usually does not create a bottleneck, because the update operations are lightweight. However, it is not recommended to use more than a few dozen or maybe hundreds of these counters as the overhead increases proportionally.

2.5 Duplicate Removal (DISTINCT)

- Duplicate removal, expressed by the DISTINCT keyword in SQL, 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: Map emits the input record as the key. A Reduce call then receives all duplicates and only needs to output the key as the representative of the group.
  - A Combiner or in-mapper combining can be applied.
  - The standard MapReduce key comparator determines, which records will be identified as identical.

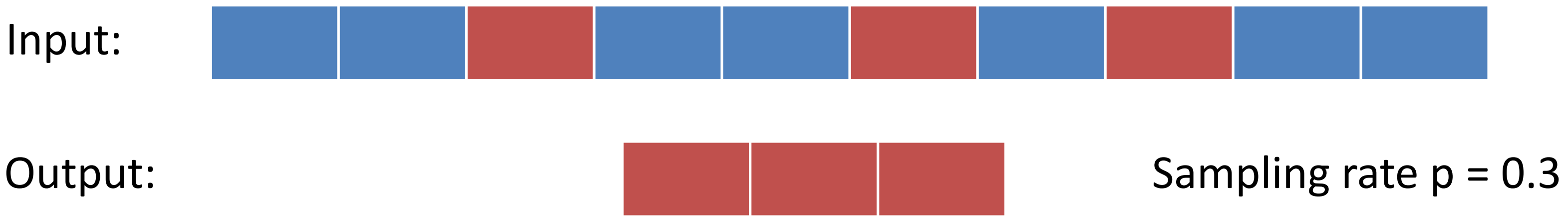
```
map( …, x )
  emit( x, NULL )
```

```
reduce( x, [NULL, NULL,…] )
  emit( x, NULL )
```

2.6 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 of a big data set are also useful for testing and debugging of MapReduce programs. It would simply take too long to test and debug on the entire big input.
- Task: Sample a fraction of approximately p, $0.0 \leq p \leq 1.0$, of the input records uniformly at random.
  - This means that each input record should have the same probability p of being selected for the output.
- Solution: A Map-only job suffices for this task. The Map function uses a pseudorandom-number generator to determine if the input record will be emitted. For instance, 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 $rnd < p$. [Link: We need to be careful when using pseudorandom-number generators.]
  - Note that for a given input set of n records, this approach will not necessarily produce exactly $p \cdot n$ output records. Due to randomness, it might produce more or fewer. In practice, when dealing with large numers of records, the resulting sample size will be very close, almost always within 5% of the ideal number.
- This approach will generate one output file per Map task. These files can be concatenated using "hadoop fs –cat." Alternatively, a single identity Reducer could perform the concatenating by emitting each record with the same key "dummy" in Map.

```
map( …, x ) {
  // Math.random() produces a number
  // rnd in the range 0.0 ≤ n < 1.0
  rnd = Math.random()

  if (rnd < p)
    emit( NULL, x )
}
```



Sampling rate p = 0.3

2.7 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 S records by simply reading the first S 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: For an input record, the Map function emits this record as the value, assigned to a key that is a random number. The Reduce function emits the values in its input list as they are read in order.
  - One can use a TotalOrderPartitioner to sort the file based on the random keys.
  - However, even the simple default hash Partitioner would suffice. Since the keys are random numbers, the hash Partitioner would assign an input record essentially to a random Reduce task. That task would output records in key order. 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.

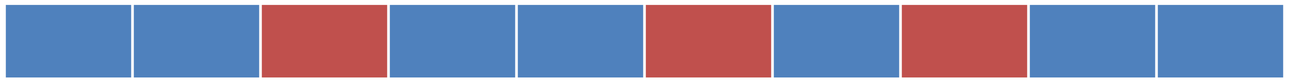
```
map( …, x ) {
 // To minimize occurrence of duplicate keys,
 // chose the random number from a large
 // domain, e.g., floating point numbers
 // between 0.0 and 1.0.
 rnd = Math.random()
 emit( rnd, x )
}
```

```
// If none of the randomly selected keys
// are identical, then the input list has
// only a single record.
reduce( rnd, [x1, x2,…] ) {
 for each x in input list
   emit( NULL, x )
}
```

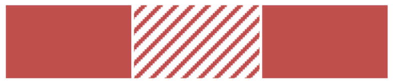
2.8 Approximate Quantiles

- Similar to minimum, maximum, and average, 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.
- Solution: The main idea is to use the Map phase to create a random sample of the input that just fits into the memory of a single machine performing the Reduce work. The Reduce function loads the data into memory, sorts it in memory, and then picks the quantiles from the sorted sample. Map assigns the same dummy key to every emitted record.
  - Since there is only a single key (the dummy key), there is only a single Reduce function call. To avoid sorting in user code, one can use the secondary sort design pattern.
- Note that if one uses secondary sort, the Reduce function does not even need to load all records from the input list into memory. If the number of records in the input list is known, it can simply scan through the (sorted!) list and pick the corresponding quantiles from the appropriate positions. If the number of records is not known, it can be determined by scanning the list. In neither case would the reduce function need more than a few bytes of memory to hold the currently read input record and maintain a counter for the position in the list.

Records sampled by Map:



Median selected from sample by Reduce:



2.8 Approximate Quantiles

- Why not use a much smaller or larger sample?
  - The smaller the sample, the less data is available for determining the quantiles in the Reducer. This leads to poorer approximation quality. On the other hand, if sample size is so big that it exceeds the amount of memory on the Reducer, then the Reduce function would have to perform more costly local disk I/O.
- Does this mean that we are now free to use a sample that exceeds the amount of memory in the Reducer?
  - Yes and no. Yes, we do not need to worry about an out-of-memory exception. On the other hand, there is no free lunch when dealing with bigger data. The MapReduce environment still has to transfer the larger sample from Mappers to Reducers and sort it by key. Hence a larger sample still results in higher cost.

```
map( …, x ) {
 // Math.random() produces a number
 // rnd in the range 0.0 ≤ n < 1.0
 rnd = Math.random()

 // Sampling rate p should be selected such
 // that p * totalInputSize is virtually guaranteed
 // to not exceed Reducer heap space.
 if (rnd < p)
   emit( dummy, x )
}
```

```
reduce( dummy, [x1, x2,…] ) {
 copy all records from the input list into array A
 sort array A

 for each quantile position i
   emit( NULL, A[i] )
}
```

2.9 Top-K Records

- In addition to quantiles, an analyst can also 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 of the records. Common examples are 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:
  - One can sort the input data and then easily select the K largest records from the sorted file. This is often the most efficient method for very large K.
  - For smaller values of K, sorting can be avoided. The main idea is to scan the input only once and use in-mapper combining to keep track of the top-K records in each Map task. A single Reduce call receives these “local” top-K lists and 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.

2.9 Top-K Records

```
Class Mapper {
 localTopK

 setup() {
  init localTopK
 }

 map(…, x ) {
  if (x is in localTopK)
   // Adding x also evicts the now
   // (k+1)-st record from localTopK
   localTopK.add( x )
 }

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

```
reduce( dummy, [x1, x2,…] ) {

 init globalTopK

 for each record x in input list
  if (x is in globalTopK)
   // Adding x also evicts the now
   // (k+1)-st record from globalTopK
   globalTopK.add( x )

 for each record x in globalTopK
  emit( NULL, x )
}
```