Intelligent Partitioning

Mirek Riedewald

Key Learning Goals

- Write the pseudo-code for the block-partitioning algorithm for frequency-counting of bird-color combinations.
- Given a partitioning for the 1-Bucket theta-join algorithm, determine input and output replication factors.
- Where is randomization used in the 1-Bucket theta-join algorithm?
- Write the pseudo-code for matrix multiplication in MapReduce, when the left matrix is partitioned horizontally and right matrix is partitioned vertically.

Introduction

- Designing a scalable algorithm requires partitioning a given big-data problem into smaller tasks. Each of these tasks should receive a small subset of the data and be able to perform its computation independently.
- How do we find such partitioning? How can we analyze its properties and reason about its performance?
- We will discuss a general strategy that starts with the most fine-grained partitioning and then groups these partitions into larger tasks. This process is driven by an optimization goal and requires well-defined measures of success. It will be explained for several example applications.

Let us start with the easiest problem.
Recall the relative frequency counting for bird species and their colors.

Reminder: “Pairs” and “Stripes”

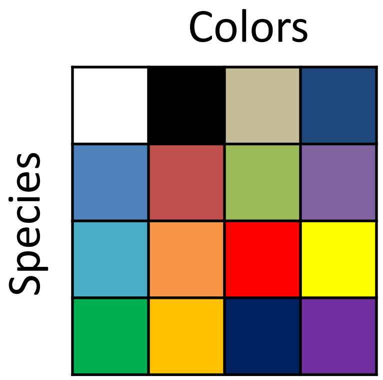
- The notion of “Pairs” versus “Stripes” surfaced in the context of the order inversion design pattern.
  - Recall the problem of estimating relative frequencies for (species, color) data records reported by citizen scientists. For each species S and color C, we wanted to compute the ratio f(S, C)/f(S), i.e., the number of (S, C) pairs divided by the total number of observations for species S.
- In the discussion below, we only consider the *frequency counting problem for f(S, C)*.

Design Space

- To identify a good problem (and data) partition scheme, we first need to identify and formalize the space of all possible partitions considered.
- For the species-color frequency counting problem, we identified species and color as possible partitioning dimensions, hence the most fine-grained partitioning was at the level of individual species-color combinations.
- Let $|S|$ and $|C|$ denote the number of different species and colors, respectively. Partitioning the given problem requires assigning each of the $|S|\cdot|C|$ different counters to some tasks responsible for computing it.
  – Pairs and Stripes are special cases for doing this.

Frequency Counting using Pairs

- We can model the space of species-color combinations as a matrix, i.e., a two-dimensional array. Each cell, representing the most fine-grained problem partitioning, corresponds to a desired count.
- The Pairs approach assigns a unique key to each individual cell. These keys can be assigned randomly to tasks.
- We show the corresponding algorithm in MapReduce below:
  - Map emits a species-color pair with count 1, aggregating the counts in Reduce. Combining should be applied if the probability of encountering the same (species, color) combination multiple times in a file split is high enough.



```
map( species S, color C )
  emit( (S, C), 1 )
```

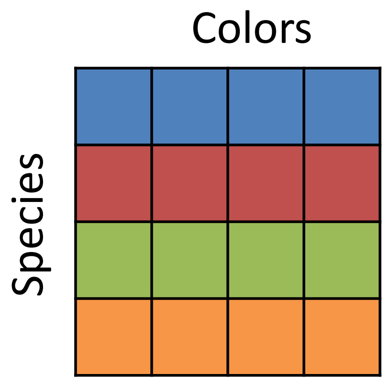
```
reduce( (S, C), [n1, n2,…] ) {
  frequency = 0

  for all n in input list do
    frequency += n

  emit( (S, C), frequency )
}
```

Frequency Counting using Stripes

- The Stripes approach groups cells in the same row together. It assigns the same key to each cell in the same row, but different keys to different rows. These keys are assigned randomly to Reduce tasks.
- To achieve stripe-based problem partitioning, the algorithm uses only the species as the key. Since each Reduce call now works with an entire row, the hash map data structure is used to keep track of the individual matrix cells in the row.



```
map( species S, color C )
  emit( S, (C, 1) )
```

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

  // H maps a color to a count
  initialize hashMap H

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

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

Comparison of Pairs versus Stripes

- **Combining**: Both approaches can use Combiners and in-mapper combining.
- **Code complexity**: The Reduce code for Pairs is simpler and easier to understand.
- **Key space**: Pairs has to deal with |S|·|C| possible intermediate keys, Stripes only with |S|. This does not affect the overhead for the application master (which depends on the number of tasks), but results in a much larger number of Reduce function calls for Pairs. On the other hand, each individual Reduce call for Stripes will be more expensive. Sorting cost during the shuffle phase might be somewhat lower for Stripes as sorting requirements are weaker.
  - Consider records (duck, red), (duck, green), and (duck, blue). For Pairs, they have to be re-ordered to (duck, blue), (duck, green), (duck, red); assuming colors are sorted alphabetically. For Stripes, the order does not matter, because key “duck” is identical for all three records.

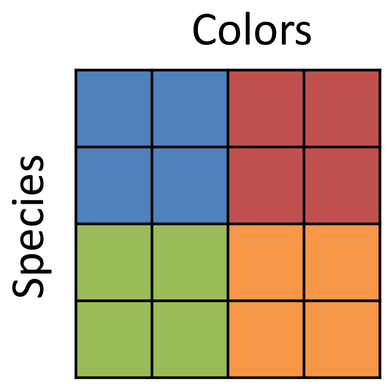
Comparison of Pairs versus Stripes (Cont.)

- **Memory**: The Map functions of both approaches do not use any local variables. Pairs’ Reduce function requires only a single variable, while Stripes’ Reduce has to maintain a data structure of size O(|C|). While not an issue for the example, in general the size of the data structure could exceed available memory, requiring more complex user code for managing it on disk.
- **Load balancing**: The more fine-grained keys in Pairs allow for greater flexibility in distributing load through use of an appropriate Partitioner. In particular, Pairs can emulate Stripes’ row-wise approach by using a Partitioner that ignores the column value of a record. On the other hand Stripes cannot emulate Pairs, in particular if Pairs assigns cells in the same row to different Reduce tasks.

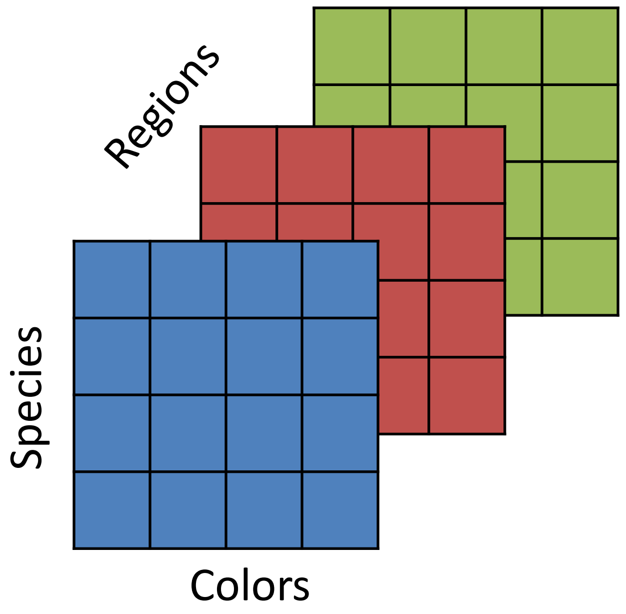
Beyond Pairs and Stripes

- Are there ways other than Pairs and Stripes to partition the frequency counting problem? Indeed, it is easy to find such strategies if we approach it as a matrix covering problem.
  - Each matrix cell corresponds to a value of interest. In the example, it is the number of occurrences of a certain species-color combination in the input data. Hence each matrix cell needs to be computed by some function call in a task. We can model this by assigning each matrix cell to exactly one of the keys. Stated differently, the matrix needs to be completely covered by r regions, each corresponding to one of the r keys.
- The best cover needs to be selected depending on data distribution and computation task.
- This approach generalizes to arrays with more than two dimensions.

Other Cover Examples



“Block”-pattern partitioning: The partitioning is based on groups of related species and colors. E.g., the top half could be birds of prey, while the bottom are other species; and the left half could be earth-tones, while the right are other colors.



Partitioning of a three-dimensional array: A “Stripe”-style approach is applied to records with three fields (species, color, region). Here each Reduce function call processes one or more regions entirely.

Evaluating a Matrix Cover

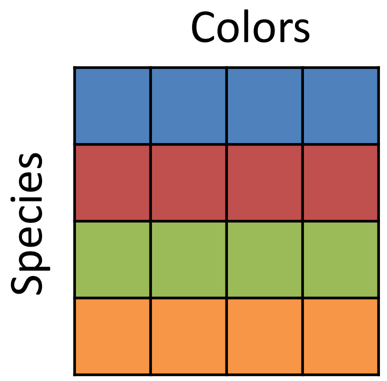
- The matrix cover directly reveals important properties of the corresponding partitioning.
- The granularity of the partitioning determines the number of distinct keys and hence Reduce function calls.
- For problems requiring data from multiple matrix cells during a computation, it is easy to check if that information will be available in a Reduce call.
  - Recall the *relative* frequency computation problem where for each bird species S and color C, the goal was to compute f(S, C) / f(S). Stripes worked very well for that problem, because both f(S) and each f(S, C) could be computed from the stripe for species S. For Pairs it was necessary to use the order inversion design pattern to pass all data needed for computing f(S) to each Reducer computing some f(S, C).

Evaluating a Matrix Cover (Cont.)

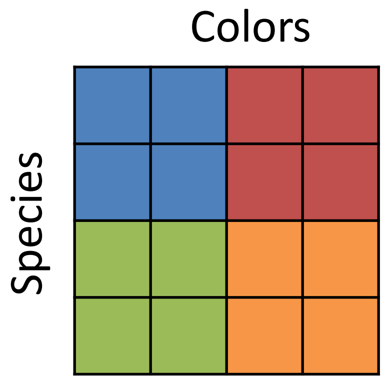
- Knowing the approximate data distribution, one can estimate how much data each partition receives. This in turn can be used to evaluate possible load balancing challenges.
  - In general, the most fine-grained partitioning, i.e., Pairs in the example, provides the greatest flexibility in assigning work evenly to Reduce tasks. For coarser partitioning schemes, e.g., Stripes in the example, one can estimate the data size assigned to each partition and determine if balanced assignment to Reduce tasks is possible.

From Matrix Cover to Algorithm

- Given a matrix cover, two challenges need to be addressed to derive the corresponding distributed algorithm:
  - Choice of keys: Each partition should have a unique key, such that all input records in that region are associated with this key.
  - Assigning keys to tasks: This is done by the Partitioner. For a random assignment, we can rely on the default hash Partitioner. If the data is very skewed, i.e., the amount of work varies significantly between different keys, one should use a more fine-grained partitioning or carefully design a custom Partitioner to balance load.
- Let us look at two MapReduce examples next.



It is trivial to achieve this partitioning by selecting species as the key.



To assign multiple species and colors to a "block" partition, the Map function needs to check if species and color of the input record fall into the corresponding range. For hierarchical attributes, this can be achieved by using a higher-order concept as the key. For instance, bird species belong to bird families, hence for records (species, family, color), one can use family as the key to create blocks of rows. As a fallback, one can use synthetic region keys:

```
map( species S, color C ) {
  if ((S = species0 OR S = species1) AND (C = color0 OR C = color1))
    emit( 0, (S,C) )       // Upper left region
  else if ((S = species0 OR S = species1) AND (C = color2 OR C = color3))
    emit( 1, (S,C) )       // Upper right region
 else if ((S = species2 OR S = species3) AND (C = color0 OR C = color1))
    emit( 2, (S,C) )       // Lower left region
 else if ((S = species2 OR S = species3) AND (C = color2 OR C = color3))
    emit( 3, (S,C) )       // Lower right region
}
```

Block Partitioning Notes

- Instead of the clumsy if-then-else statements, the regions can be encoded more elegantly as a sequence of ranges in each dimension. Binary search on those ranges will efficiently determine all relevant region identifiers.
  - Recall the discussion of range partitioning for sorting, and the TotalOrderPartitioner in MapReduce in particular.
- This block partitioning addresses the problem of choosing the right dummy colors for the order inversion solution, when we want to split on the color dimension.
  - We create one dummy color for every block, i.e., for each input record (S, C), we also assign (S, dummy_i) to each block i.

Partitioning in Spark

- (Pair) RDDs also support custom Partitioners, hence the algorithms carry over from MapReduce.
- DataSet and DataFrame currently do not support custom Partitioners, leaving this choice to an automatic optimizer.
  - What can we do here as a programmer trying to control partitioning? Using map(), we can assign custom keys to the rows of a DataSet, encoding regions of a partitioning. Then groupBy() on that key column, together with the appropriate “aggregate” function can process the region.

Now we move on to a more challenging operation: the theta-join.

Theta-Joins

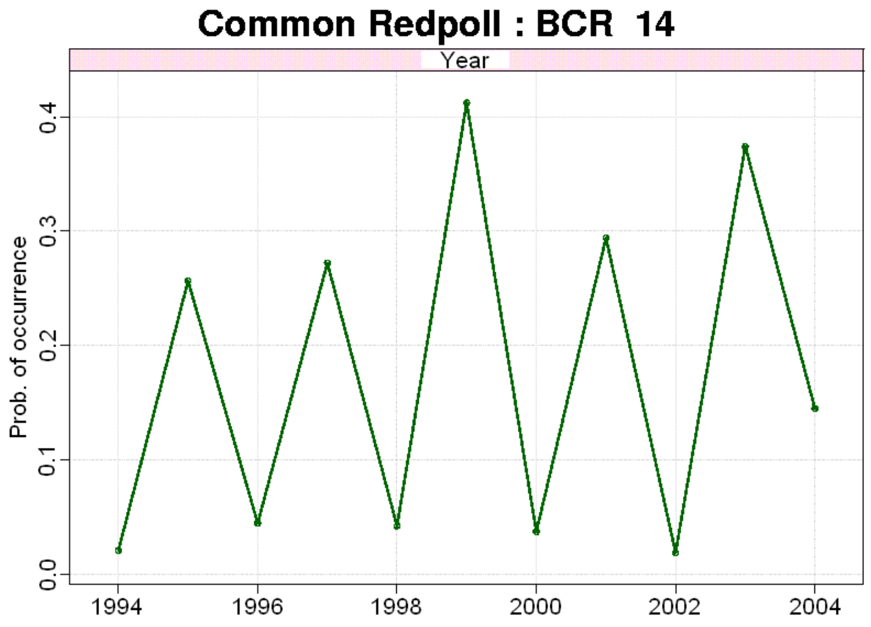
- The idea of modeling the partitioning of a big-data problem as a matrix or array covering problem is very general. To illustrate this point, we show how it can be applied to theta-joins. Here all region keys are synthetic and have nothing to do with the values occurring in input tuples.
- Despite the same basic idea of matrix covering, the different nature of the join problem will result in a somewhat different analysis of the resulting algorithms.
- This discussion is based on a paper published by Prof. Riedewald's research group
  - A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In Proc. ACM SIGMOD Int. Conf. on Managament of Data, pages 949-960, 2011

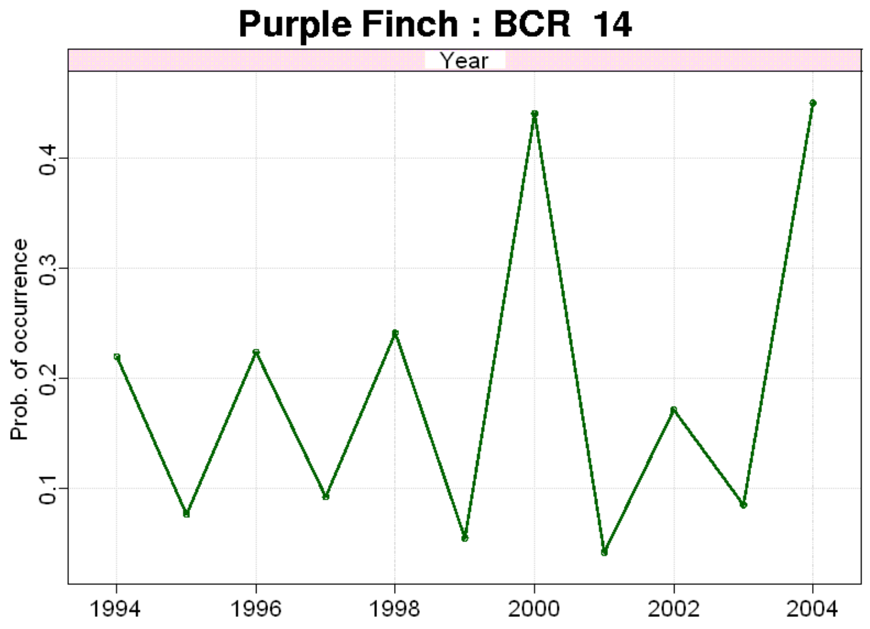
Problem Definition

- In an earlier module, we already discussed distributed algorithms for equi-joins. An equi-join is a special type of theta-join. A general theta-join is defined as follows:
  - Given two data sets S={s_1, s_2,…} and T={t_1, t_2,...}, find all pairs (s_i, t_j) that satisfy some predicate P(s_i, t_j) over the values of the attributes of the S-tuple and T-tuple.
- One of the most common types of non-equi theta-joins are those with inequality conditions. For example, two weather data sets could be joined to find pairs of observation records that are "near" each other in space and time. Similarly, an ornithologist might be interested in finding bird species with similar or opposite trends in some region.
- Our goal is to partition any given theta-join computation problem such that job completion time on a given number of worker machines is minimized. (Alternative goals, e.g., minimizing total cost, are not considered here.)

Theta-Join Example: Habitat Competition

- Assume ornithologists created a large database of summaries showing the association between some variable(s) of interest, e.g., the year, and the probability of observing a species in some region. A *pair* of such summaries is of interest, if (1) the summaries are for different species, (2) they cover the same region and variable of interest (year in the example), and (3) the trends are very different.
  - In the example, both species show an interesting bi-annual trend, with one species peaking in even years and the other in odd years.
- Such pairs of related summaries help the ornithologists discover hypotheses about potential habitat competition.



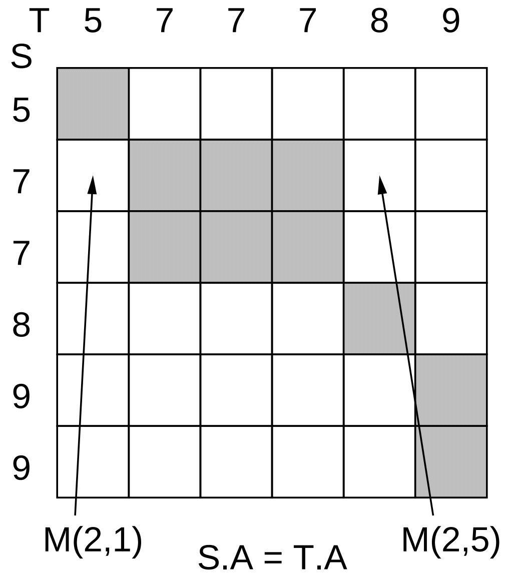


Challenges

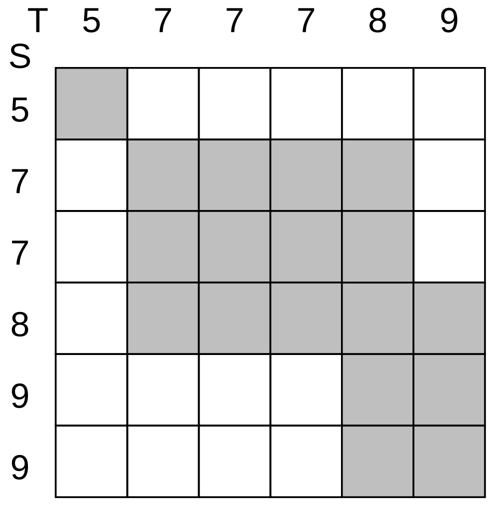
- Recall the hash+shuffle and partition+broadcast algorithms.
  - Partition+broadcast conceptually checks each S-tuple against the entire set T (which is broadcast to all tasks) to find the matching pairs (s_i, t_j). Hence it can be used to implement any theta-join. Unfortunately, it will only be efficient if T is small—ideally small enough to fit in memory.
  - Hash+shuffle works well even if both inputs are big. However, (1) it does not generalize to non-equi joins and (2) it does not scale if the join attribute's domain is small or shows a highly skewed distribution.
- To be able to deal with arbitrary theta-joins, we need an algorithm that works for big data and supports any possible join predicate.
- To minimize job completion time, we need to partition the join computation problem such that we minimize the amount of work assigned to the machine with the most work. (This machine determines the end of the job!) This partitioning should work well, even if the data is skewed.
- The matrix covering approach will prove extremely useful for reasoning about this problem and for finding a solution.

Theta-Join Matrix

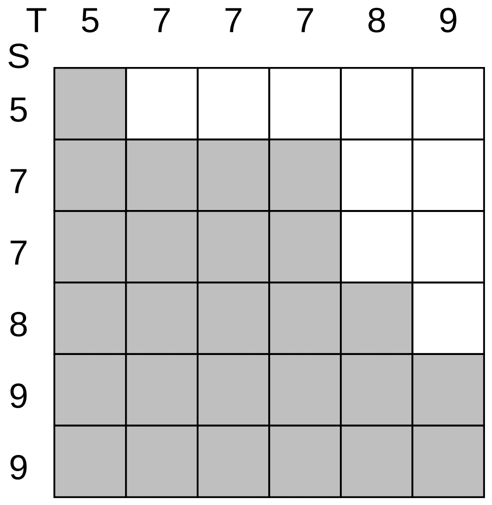
- Recall that any theta-join result is a subset of the cross-product S×T, which combines each tuple from S with each tuple from T. Hence any theta-join can be represented by a matrix M with |S| rows (one for each S-tuple) and |T| columns (one for each T-tuple).
- Matrix cell M(i, j) corresponds to the pair (s_i, t_j). Its value is “true”, if (s_i, t_j) satisfies the join predicate, i.e., is in the join result; and “false” otherwise.



S.A = T.A



abs(S.A - T.A) < 2



S.A >= T.A

Discussion of Example Matrices

- In the above example, S and T have a column named A and values S={5, 7, 7, 8, 9, 9} and T={5, 7, 7, 7, 8, 9}. (Note that both are multi-sets, i.e., might contain the same tuple multiple times.)
- The left matrix represents the result of equi-join S.A = T.A. Matrix entry M(2,1) corresponds to the pair (7, 5), which are the second S-tuple and the first T-tuple, respectively. Since 7 is not equal to 5, the pair does not belong to the join result and hence the matrix cell's value is "false." Cells with value "false" are not shaded, while those with value "true" are shaded. For instance, M(1,1) in the upper left corner corresponds to matching pair (5, 5) and hence is shaded.
- The example in the center represents a band-join searching for pairs with similar values.
- The example on the right shows a join with an inequality predicate.

Join Matrix in Practice

- Note that the join matrix as discussed so far encodes the exact join result.
  - If that matrix was available from the start, then we would not need to execute the join—the result is already in the matrix.
- In practice, the join matrix is not given to us. Instead, we have to rely on cheaper algorithms, e.g., approximate quantiles and output samples, to estimate the amount of input and output for a region of the matrix.

Matrix Cover

- For correctness, join matrix M needs to be covered by regions corresponding to keys. If a matrix cell c is covered by the region corresponding to key K, we will say that “key K covers c” or that the corresponding Reduce call covers c.
  - We use MapReduce terminology here to simplify the discussion. The same ideas extend to Spark.
- Each join result tuple should be computed by some Reduce call, therefore every “true”-valued matrix cell has to be covered by some key.
  - Again, in practice we do not have the exact locations of all true-values cells. Instead, we must identify regions of candidate cells that are guaranteed to include all output tuples.
- “False”-valued cells do not need to be covered. On the other hand, covering them does not jeopardize algorithm correctness: The Reduce call processing that key can easily reject the corresponding tuple-pair by checking the join condition for it.
  - For example, a Reduce call processing M(2,1) is responsible for exploring pair (s_2=7, t_1=5). For join predicate S.A = T.A, this pair would not be output; but for predicate S.A >= T.A, it would.

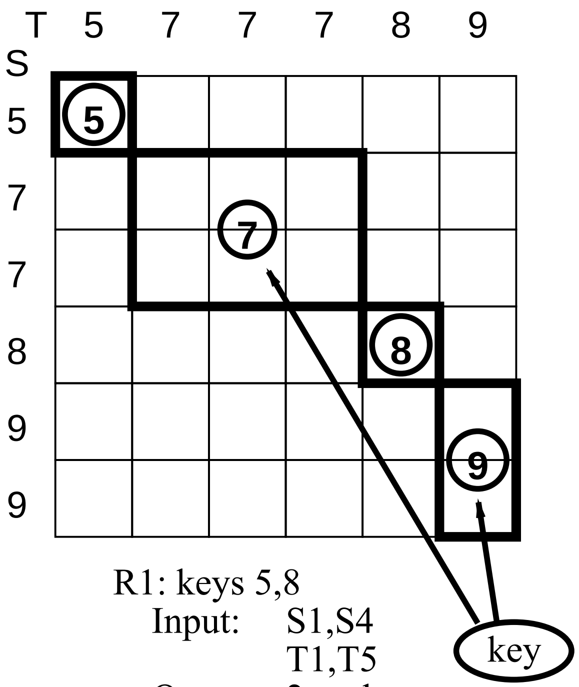
Matrix Cover (Cont.)

- No matrix cell should be covered by multiple keys, because this would result in “undesirable” output duplicates.
  - Consider tuples s_2=7, s_3=7, and t_2=7. Result tuples (s_2, t_2) = (7, 7) and (s_3, t_2) = (7, 7) look the same, but are not “undesirable duplicates”, because they are produced by two distinct S-tuples. On the other hand, producing (s_2, t_2) more than once would be undesirable. (The result would contain additional (7, 7) tuples that should not be there.)
- Undesirable duplicates could be removed in a post-processing phase, but there are drawbacks:
  - The additional phase is expensive.
  - To determine if two identical output tuples are “undesirable,” additional provenance information has to be attached to each output tuple, increasing output size.
  - Operators processing the join output, e.g., to count the number of output tuples, cannot be pushed into the Reduce phase of the join.

Matrix Cover (Cont.)

- In general, there will be many possible matrix covers that satisfy the above conditions. To select the winner, we have to analyze their properties.
- For most join problems it is reasonable to assume that the more input tuples a worker has to process and the more output tuples it generates, the longer it will run. To minimize job completion time, we therefore are interested in minimizing max-input or max-output. The former is the size of the largest input any of the worker machines receives; the latter is the size of the largest output any worker produces.
- For the MapReduce implementation, we will assign exactly one Reduce task to each worker. Hence our goal is to minimize max-reducer-input or max-reducer-output, i.e., the largest input and output, respectively, any Reduce task is responsible for.
- Let us look at example covers to better understand the tradeoffs.

Hash+shuffle Join:



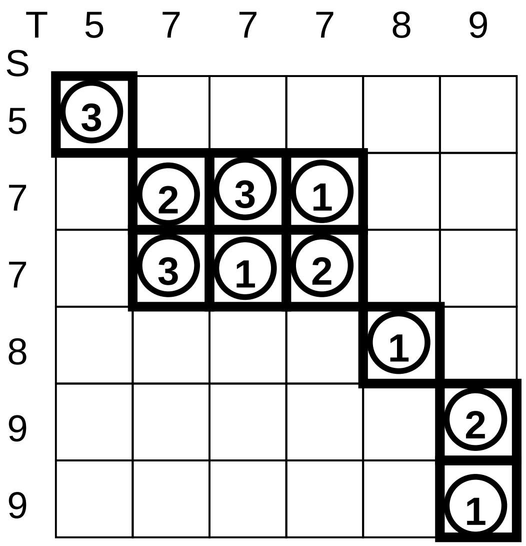
R1: keys 5,8
Input: S1,S4
T1,T5
Output: 2 tuples

R2: key 7
Input: S2,S3
T2,T3,T4
Output: 6 tuples

R3: key 9
Input: S5,S6
T6
Output: 2 tuples

max-reducer-input = 5
max-reducer-output = 6

Random Assignment:



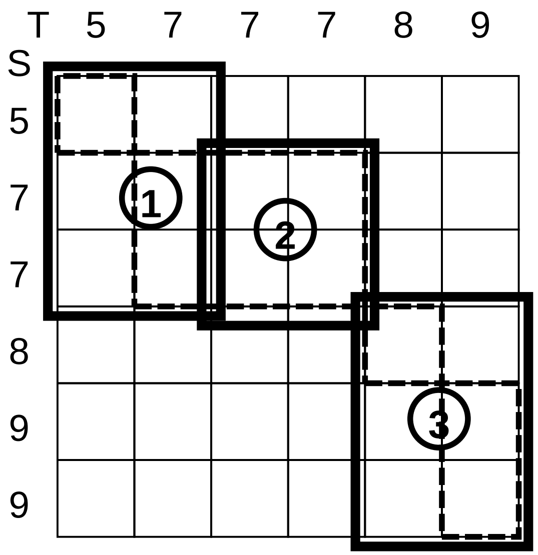
R1: key 1
Input: S2,S3,S4,S6
T3,T4,T5,T6
Output: 4 tuples

R2: key 2
Input: S2,S3,S5
T2,T4,T6
Output: 3 tuples

R3: key 3
Input: S1,S2,S3
T1,T2,T3
Output: 3 tuples

max-reducer-input = 8
max-reducer-output = 4

Balanced Algorithm:



R1: key 1
Input: S1,S2,S3
T1,T2
Output: 3 tuples

R2: key 2
Input: S2,S3
T3,T4
Output: 4 tuples

R3: key 3
Input: S4,S5,S6
T5,T6
Output: 3 tuples

max-reducer-input = 5
max-reducer-output = 4

Explanation of The Example

- The first cover corresponds to hash+shuffle, using the join attribute as key and resulting in 4 distinct regions. To create three Reduce tasks, let keys 5 and 8 be assigned to Reducer R1, key 7 to R2, and key 9 to R3. R2 is responsible for generating the 6 output tuples corresponding to the combinations of 3 tuples with value 7 from S and 2 such tuples from T. Hence R2 receives 5 input tuples and produces 6 output tuples; the others receive significantly less work. The load imbalance is caused by the data skew, in particular value 7.
- To address the skew problem, we can try and assign keys randomly to the "true"-valued cells in the matrix. This lowers max-reducer-output to 4, but increases max-reducer-input. For example, input tuple s_2=7 has to be sent to all three Reducers, because it is needed to compute cells marked with keys 1, 2, and 3 in the second row. It would also be challenging to implement this cover in practice. Consider Reducer R3, which receives s_2=7, s_3=7, t_2=7, and t_3=7. Without careful coordination with R1 and R2, R3 might also produce the output corresponding to M(2,2) (which is assigned to R2) and M(3,3) (which is assigned to R1).
- The best overall solution is shown in the third cover. Even though some "false"-valued cells are covered unnecessarily, the cover achieves max-reducer-input as low as hash+shuffle, and at the same time max-reducer-output as low as the random assignment. We want to find such covers.

From Matrix Cover to Algorithm

- Before discussing how to find optimal matrix covers, we first have to establish how such a cover can be converted into a distributed algorithm for computing the theta-join.
- It turns out that *if we cover the entire matrix*, then a simple randomized 2-round algorithm can achieve this goal.
  - In round 1, each task independently assigns the region identifiers, possibly replicating input tuples if necessary.
  - The data is shuffled to group it by region. Then tasks in round 2 compute the join in each region separately. This local computation can leverage existing libraries of efficient (non-parallel) join implementations, e.g., index-based approaches for equi-join and inequality joins.
- We will refer to this algorithm as Basic-Theta. Its MapReduce implementation is discussed next.

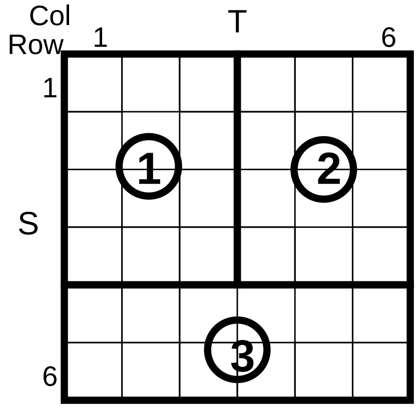
```
Class Mapper {
 // A Cover of the entire join matrix
 Cover

 setup() {
  Cover = load covering information from file cache
 }

 map( tuple x ) {
  if (x is from S) {
   // Select a random row of the matrix
   matrixRow = random( 1, |S| )

   // Find all regions intersecting this row and emit
   // tuple x with the corresponding key
   for each regionID in Cover.getRegions( matrixRow )
    emit( regionID, (x, "S") )
  }
  else {    // x is from T
   // Select a random column of the matrix
   matrixCol = random( 1, |T| )

   // Find all regions intersecting this column and emit
   // tuple x with the corresponding key
   for each regionID in Cover.getRegions( matrixCol )
    emit( regionID, (x, "T") )
  }
 }
}
```



Example covering

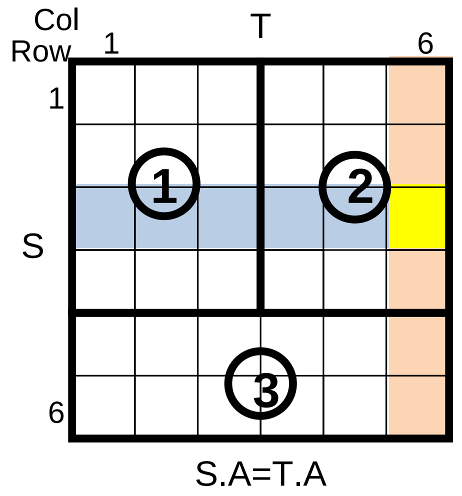
```
reduce( regionID, [(x1, flag1), (x2, flag2),…]) {
 initialize S_list and T_list

 // Separate the input list by the data set the tuples came from
 for all (x, flag) in input list do
   if (flag = “S”)
     S_list.add( x )
   else
     T_list.add( x )

 // Any appropriate (non-parallel) join implementation can be used to join S_list and T_list
 joinResult = myFavoriteJoinAlgorithm( S_list, T_list)
 for each tuple t in joinResult
   emit( t )
}
```

Algorithm Correctness

- As long as the given matrix cover guarantees that each cell is covered by *exactly* one key, it is easy to show that the algorithm correctly implements any given theta-join.
- Consider a join result tuple (s_i, t_j). Tuples s_i and t_j are assigned to a random row and column, respectively. This row and column intersect in exactly one matrix cell, which is covered by exactly one key. This Reduce call receives both tuples and can compute the result pair. Since no other Reduce call receives both tuples, this result will not be produced anywhere else.
  - The example illustrates this argument for tuples s_1 and t_1. The randomly selected row and column intersect in region 2, hence the Reduce call for key 2 produces the output.



Map:

Input tuple	Random row/col	Output
S1.A=5	3	(1,S1),(2,S1)
S2.A=7	5	(3,S2)
S3.A=7	1	(1,S3),(2,S3)
S4.A=8	5	(3,S4)
S5.A=9	1	(1,S5),(2,S5)
S6.A=9	2	(1,S6),(2,S6)
T1.A=5	6	(2,T1),(3,T1)
T2.A=7	2	(1,T2),(3,T2)
T3.A=7	2	(1,T3),(3,T3)
T4.A=7	3	(1,T4),(3,T4)
T5.A=8	6	(2,T5),(3,T5)
T6.A=9	4	(2,T6),(3,T6)

Reduce:

Reducer X: key 1
Input: S1, S3, S5 ,S6
T2, T3, T4
Output: (S3,T2),(S3,T3),(S3,T4)

Reducer Y: key 2
Input: S1, S3, S5, S6
T1, T5, T6
Output: (S1,T1),(S5,T6),(S6,T6)

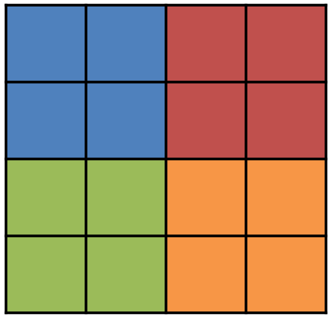
Reducer Z: key 3
Input: S2, S4
T1, T2, T3, T4, T5, T6
Output: (S2,T2),(S2,T3),
(S2,T4),(S4,T5)

Implementation in Spark

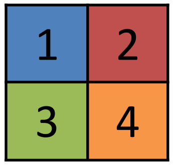
- The matrix cover can be broadcast, or passed by the driver to all tasks. In practice matrix cover information is small.
- For pair RDD implementation, a flatMap call creates the tuple duplicates and adds the region ID as the key; processing each input RDD separately.
- Then aggregateByKey produces pair RDDs with schema (regionID, listOfRecordsInRegion) for each input separately.
- Finally we join the two grouped inputs on regionID and map the value component to the join result in the region.
  - Note that the regionID key is associated with a pair of lists—the input tuples from left and right input, respectively, in the region.
- The DataSet program is analogous.

Why Randomization?

- It might come as a surprise that each S-tuple s_i is assigned to a random row, instead of "correct" row i. (The same applies to T-tuples and columns.) Randomization is a powerful tool for two reasons:
- Simpler algorithm:
  - Mappers do not know the "correct" row of an input tuple. For example, tuple $s_4 = 8$ could belong to row 1 if there are no smaller values; or it could be in row 6 if it is the greatest value. A pre-processing step would have to add row and column numbers.
  - By removing the one-to-one correspondence between input tuples and rows/columns, the join matrix does not need to have |S| by |T| cells and could be much smaller as shown below.
- Performance:
  - Avoiding the pre-processing phase obviously reduces cost.
  - Randomization effectively removes output skew. As the examples illustrated, "true"-valued cells could be clustered in some region of the join matrix. A key covering such a dense region would receive an unfairly large share of the output. Randomizing rows and columns effectively shuffles the "true"-valued cells around in the matrix, balancing load evenly in the system.
  - Even though randomization could in theory result in poor load distribution, e.g., if all S-tuples are randomly assigned to the same row, the probability of this to happen is very low in practice, especially for big data. Using Chernoff bounds, we can show that for big data the probability of a Reduce call receiving 5% or more over its target input share is virtually zero.



Both matrices describe the same covering: each of the four Reduce calls receives 50% of S and 50% of T. The right matrix expresses this with one fourth the number of cells compared to the one on the left.



Optimal Cover: Entire Matrix

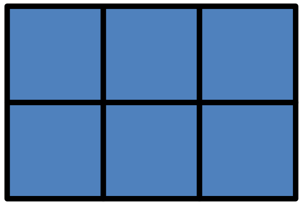
- We established earlier that any covering of the entire join matrix with non-overlapping regions guarantees correctness for any theta-join implementation.
- However, there are many possible non-overlapping matrix covers. Given r, the desired number of regions, our goal is to find the optimal cover.
- We will demonstrate the power of the matrix covering approach by discussing how it allows us to derive lower bounds. Then we introduce the 1-Bucket-Random algorithm. This algorithm often comes close to the lower bounds and only needs minimal information about the input data. Hence it is often the best possible solution for computing a given theta-join.

Ideal Cover: Use Squares

- **Lemma 1**: A region that covers c cells of join matrix M will receive at least $2\sqrt{c}$ input tuples.
- Proof:
  - Consider a region receiving x S-tuples $\{s_1, s_2, \ldots, s_x\}$ and y T-tuples $\{t_1, t_2, \ldots, t_y\}$. This region can cover at most $x \cdot y$ matrix cells, corresponding to all combinations (x_i, y_j), $1 \leq i \leq x$, $1 \leq j \leq y$. To cover c cells, it has to hold that $x \cdot y \geq c$.
  - From $(x-y)^2 \geq 0$ and $x \cdot y \geq c$, we derive $x^2+y^2 \geq 2xy \geq 2c$.
  - Now consider $(x+y)^2 = x^2+y^2+2xy$. From the above follows that $(x+y)^2 \geq 2c+2c=4c$, therefore $x + y \geq 2\sqrt{c}$.
- Note that if c is a perfect square, then a square-shaped region of $\sqrt{c}$ by $\sqrt{c}$ rows and columns matches the lower bound established by the lemma.

Lower Bounds: Cross-Product

- Consider computing the full cross-product S×T, i.e., every matrix cell belongs to the output.
- We assign exactly one key to each Reduce task. (One can show that assigning multiple keys to a Reduce task results in unnecessary duplication of input tuples.)
- **Max-reducer-output**: The matrix consists of |S|·|T| cells. Max-reducer-output is minimized if each of the r keys covers the same number of cells, i.e., |S|·|T|/r.
- **Max-reducer-input**: Lemma 1 implies that covering |S|·|T|/r matrix cells requires sending at least $2\sqrt{|S| \cdot |T|/r}$ input tuples to the Reducer.
- Is it possible to match these lower bounds in practice?
  - Yes, depending on the values of |S|, |T|, and r. For example, let |S|=4,000,000, T=6,000,000, and r=6. Each key would cover a square of 2,000,000 by 2,000,000 cells, corresponding to a 2-by-3 grid partitioning of the matrix. Unfortunately, things do not work out this well for other combinations of |S|, |T|, and r, e.g., |S|=3,000,000, |T|=8,000,000, and r=6.
- This raises the question of how close we can get to the lower bounds in practice for any given combination of |S|, |T|, and r.

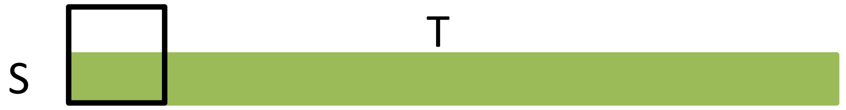


Partitioning matching the lower bound for |S|=4,000,000, |T|=6,000,000, and r=6.

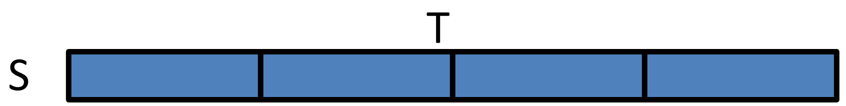
Extreme Case: Small-vs-Large

- Consider the extreme case where one input set is more than r times bigger than the other. Without loss of generality, let |S| < |T|/r. Even though the lower bounds for max-reducer-input and max-reducer-output cannot be matched in this case, it is easy to show that the optimal partitioning consists of regions of |S| rows by |T|/r columns.
- Interestingly, this partitioning corresponds to the partition+broadcast algorithm: The entire input S is broadcast to all workers, who join it with a partition of the larger input T.

"Idealistic" square region



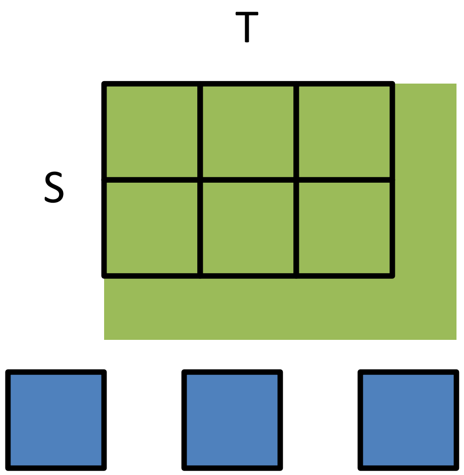
Actual optimal region



Non-Extreme Case

- Consider the remaining case where |T|/r ≤ |S| ≤ |T|.
  - This implies $\sqrt{|S||T|/r} \leq |S| \leq |T|$.
- For ease of exposition, let K=$\sqrt{|S||T|/r}$ for this discussion.
- For some combinations of |S|, |T|, and r it is not possible to cover the matrix with r regions of K by K cells.

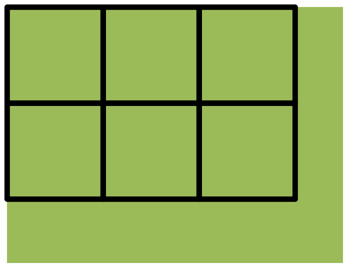
Ideal square region of size K by K



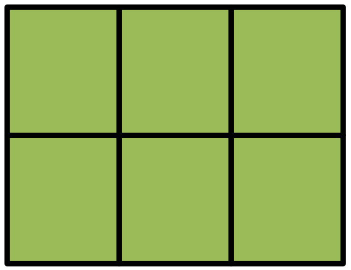
In this example for r=9, only six K-by-K squares fit into the matrix. The remaining three squares together contain as many cells as the “leftover” of the matrix, but cannot cover this leftover due to its shape.

Non-Extreme Case (Cont.)

- Whenever the matrix cannot be covered by r K-by-K squares, we conceptually "inflate" the regions so that a complete cover is achieved.
- This approach essentially wastes some of the regions (those Reduce tasks receive no data), while assigning more work to the others. (Larger regions imply more input and output for those Reducers.)
- Despite this "waste", one can still prove good properties of the algorithm.


Only 6 of the 9 K-by-K squares "fit".

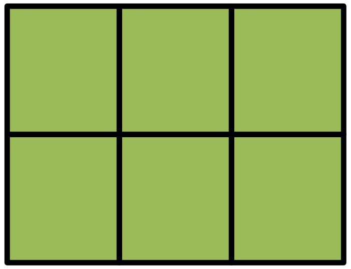
"Wasted" regions.


The "inflated" 6 regions cover the matrix.

Non-Extreme Case (Cont.)

- Formally, the cover with inflated regions is created by partitioning the join matrix into a regular grid of $\mathrm{A} = \left\lfloor \sqrt{\frac{|S|}{|T|}} r \right\rfloor$ by $\mathrm{B} = \left\lfloor \sqrt{\frac{|T|}{|S|}} r \right\rfloor$ identical regions.
- It is easy to show that A·B ≤ r, therefore this cover does not use more than the desired number of regions.
- But how much worse are max-reducer-input and max-reducer-output for this cover? After all, it uses potentially fewer than r regions, and each of them is inflated to a larger size.

Assume that r = 9 and that T has 50% more tuples than S, i.e., |S|/|T|=2/3. Then $A = \lfloor \sqrt{6} \rfloor = 2$ and $\mathrm{B} = \lfloor \sqrt{13.5} \rfloor = 3$. Hence the join matrix is partitioned into 2 by 3 regions, each of size |S|/2 by |T|/3.



Non-Extreme Case (Cont.)

- **Upper bound for max-reducer-output**: Recall the lower bound |S||T|/r.
  - Each partition consists of |S|/ $\left\lfloor \sqrt{\frac{|S|}{|T|}r} \right\rfloor$ by |T|/ $\left\lfloor \sqrt{\frac{|T|}{|S|}r} \right\rfloor$ cells.
  - Since $A = \left\lfloor \sqrt{\frac{|S|}{|T|}r} \right\rfloor$ is obtained by rounding down $\sqrt{\frac{|S|}{|T|}r}$, it holds that $\left\lfloor \sqrt{\frac{|S|}{|T|}r} \right\rfloor > \frac{A}{A+1}\sqrt{\frac{|S|}{|T|}r}$. This in turn implies that $\frac{|S|}{\left\lfloor \sqrt{\frac{|S|}{|T|}r} \right\rfloor} < \frac{|S|}{\frac{A}{A+1}\sqrt{\frac{|S|}{|T|}r}} = \frac{A+1}{A}\sqrt{|S||T|/r}$. We can show similarly that $\frac{|T|}{\left\lfloor \sqrt{\frac{|T|}{|S|}r} \right\rfloor} < \frac{B+1}{B}\sqrt{|S||T|/r}$.
  - Hence the number of cells covered by an inflated region is less than $\left(\frac{A+1}{A}\sqrt{|S||T|/r}\right)\left(\frac{B+1}{B}\sqrt{|S||T|/r}\right) = \frac{(A+1)(B+1)}{A \cdot B}|S||T|/r$.
  - Since both A and B are at least 1 (by definition for the non-extreme case), it follows that (A+1)(B+1)/(A·B) ≤ 4. Stated differently, the upper bound of max-reducer-input for the partitioning with inflated regions is less than 4 times the lower bound.
  - For greater r, the values of A and B tend to become greater as well. This further improves the inequality. For example, for A, B ≥ 10, it holds that (A+1)(B+1)/(A·B) ≤ 1.21. Then the upper bound of max-reducer-output is within 21% of the lower bound! This is good news, because in practice r tends to be fairly large in order to use more machines.

Non-Extreme Case (Cont.)

- **Upper bound for max-reducer-input**: Recall the lower bound $2\sqrt{|S||T|/r}$.
  - From the analysis above also follows that each region receives less than $\left(\frac{A+1}{A}\sqrt{|S||T|/r}\right) + \left(\frac{B+1}{B}\sqrt{|S||T|/r}\right) = (2 + \frac{1}{A} + \frac{1}{B})\sqrt{|S||T|/r}$ input tuples. Stated differently, it receives less than twice the lower bound. For larger r, and hence larger A and B, the upper bound will be very close to the lower bound.

1-Bucket-Random

- The 1-Bucket-Random algorithm, or 1-Bucket for short, is based on the analytical results on lower and upper bounds. For the results to apply, it sets *r to the number of Reduce worker machines available and assigns a single key to each Reduce task*.
- 1-Bucket-Random can implement any theta-join and it requires only minimal statistics. In particular, knowing the size of S and T is sufficient (corresponding to a histogram about the frequency distribution of values in a data set that has only a single bucket). In fact, it suffices to know the ratio $|S|/|T|$.
  - Assume without loss of generality that $|S|\leq|T|$ and let $C = |S|/|T|$.
  - If $C < 1/r$, then set A = 1 and B = r.
  - Otherwise, i.e., if $C \geq 1/r$, set $\mathrm{A} = \left\lfloor\sqrt{C \cdot r}\right\rfloor$ and $\mathrm{B} = \left\lfloor\sqrt{C^{-1} \cdot r}\right\rfloor$.

1-Bucket-Random: Map

```
map(…, tuple x) {
 if (x is from S) {
  // Select a random integer from range [0,…, A-1]
  row = random( 0, A-1 )

  // Emit the tuple for all regions in the selected "row".
  for key = (row * B) to (row * B + B – 1)
   emit( key, (x, "S") )
 }
 else {    // x is from T
  // Select a random integer from range [0,…, B-1]
  col = random( 0, B-1 )

  // Emit the tuple for all regions in the selected "column".
  // This requires skipping B region numbers forward from
  // start region key equal to col.
  for key = col to ((A-1)*B + col) step B
   emit( key, (x, "T") )
 }
}
```

0	1	2
3	4	5

Partitioning for A=2 and B=3. The numbers indicate the region keys.

Note that Map does not need the matrix cover any more. It can simply compute the region numbers on-the-fly from r and |S|/|T|.

1-Bucket-Random: Reduce

```
reduce( regionID, [(x1, flag1), (x2, flag2),…]) {
  initialize S_list and T_list

  // Separate the input list by the data set the
  // tuples came from
  for all (x, flag) in input list do
    if (flag = "S")
      S_list.add( x )
    else
      T_list.add( x )

  // Any appropriate (non-parallel) join implementation
  // can be used to join S_list and T_list
  joinResult = myFavoriteJoinAlgorithm( S_list, T_list)
  for each tuple t in joinResult
    emit( t )
}
```

This reduce function is identical to Basic-Theta, the generic version of the algorithm shown earlier for arbitrary matrix cover.

Implementation in Spark

- This is identical to the version presented earlier for Basic-Theta. We only have to change the function assigning keys to input duplicates in the flatMap.
  - Instead of an actual matrix cover, only values A and B need to be passed to all tasks.
- For pair RDD implementation, a flatMap call creates the tuple duplicates and adds the region ID as the key; processing each input RDD separately.
- Then aggregateByKey produces pair RDDs with schema (regionID, listOfRecordsInRegion) for each input separately.
- Finally we join the two grouped inputs on regionID and map the value component to the join result in the region.
  - Note that the regionID key is associated with a pair of lists—the input tuples from left and right input, respectively, in the region.
- The DataSet program is analogous.

1-Bucket-Random Analysis: Cross-Product

- 1-Bucket-Random relies on the matrix cover used in the analysis of lower and upper bounds. The analytical results guarantee that each Reduce call receives close to the optimal amount of input and is responsible for producing close to the optimal amount of output. By assigning exactly one key to each Reduce task and exactly one Reduce task to each worker, these guarantees extend to max-input and max-output for the worker machines.
- The guarantees become stronger for larger r. Hence for typical big data applications running on 100 or more machines, 1-Bucket-Random distributes work in a near-optimal manner.
  - These guarantees are probabilistic, due to the random assignment of input tuples to matrix rows and columns. However, for big data, the probability of a “bad” assignment is virtually zero.
- While 1-Bucket-Random achieves near-optimal work assignment for the cross-product, many joins in practice compute a much smaller output. How well does the algorithm perform for those joins?

1-Bucket-Random Analysis: Output-Dominated Joins

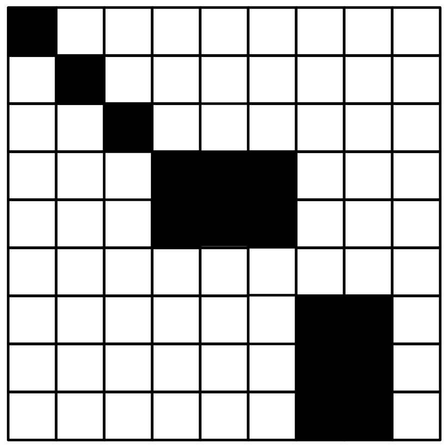
- First consider joins where the output is much larger than the input, e.g., 100 times larger or more.
- For these joins, the time for producing the output and writing it to the distributed file system dominates the total job execution time. Hence one has to minimize max-reducer-output.
- The analytical results show that each region of the matrix cover contains a near-optimal number of cells. Since input tuples are randomly assigned to matrix rows and columns, this implies that on expectation each of these regions will produce a near-optimal number of output tuples. Experimental results show that this is indeed the case in practice.
  - Stated differently, for output-dominated joins, 1-Bucket-Random will also achieve a near-optimal assignment of work.

1-Bucket-Random Analysis: Input-Dominated Joins

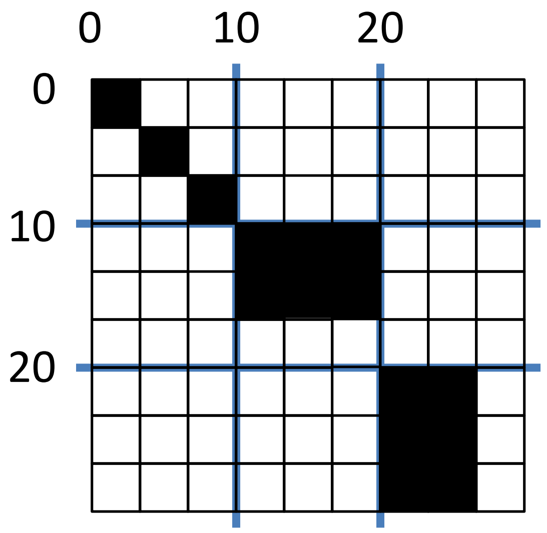
- These are joins where the output is smaller or not much larger than the input.
- For these joins, 1-Bucket-Random will also perform a near-optimal assignment of output-related work. However, this is irrelevant because input-related costs dominate. Hence one has to focus on max-reducer-input.
- This reveals the weakness of 1-Bucket-Random, because it has to send each S-tuple to B tasks, and each T-tuple to A tasks. The resulting input duplication further increases the already dominant input-related costs for shuffling and local processing in the Reducers.
- Why does the previous upper and lower bound analysis for max-reducer-input not apply here any more? The upper bound still applies as before, but the lower bound does not.
  - Recall that only the true-valued cells of the join matrix have to be covered. By not covering some cells of the matrix, smaller regions can be used, resulting in a smaller lower bound for input duplication.
- Intuitively, for input-dominated joins, one can improve over 1-Bucket-Random by not covering “empty” regions of the matrix, i.e., regions that cannot contain any results.

Improved Algorithms for Input-Dominated Joins

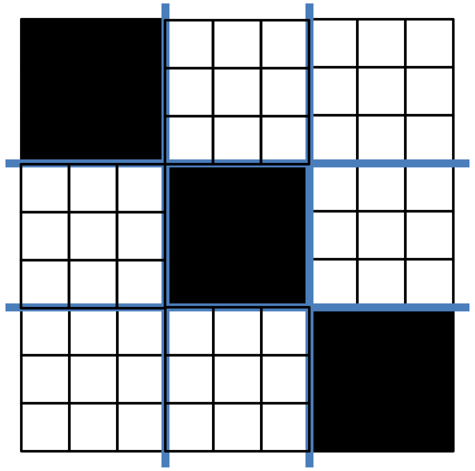
- To avoid covering some region of the join matrix, one has to prove that no cell in that region corresponds to a join result tuple. This requires knowledge about the data properties in that region.
- For example, consider an equi-join of S and T on attribute A, using condition S.A = T.A. Assume A takes on values between 0 (inclusive) and 30 (exclusive). One can partition A's domain into ranges, for instance [0,10), [10, 20), and [20, 30). Clearly, an S-tuple in range [10,20) cannot join with a T-tuple in range [20,30). Based on this analysis, most regions of the join matrix do not need to be covered, greatly reducing the size of each of the r regions, and hence Reducer input.
- We next illustrate this idea with an example.



Join matrix for equi-join S.A = T.A.



Range-partitioned matrix. The domain of join attribute A is partitioned into three ranges: [0,10), [10, 20), and [20, 30). Each S-tuple falls into exactly one row range; each T-tuple falls into exactly one column range.



Candidate cells that have to be covered. The equality condition prevents regions with non-intersecting A-ranges for S and T from containing any "true"-valued cells. Hence only the shaded regions need to be covered. Not all cells in these regions will produce join output, but this cannot be determined from the partitioning into three ranges.

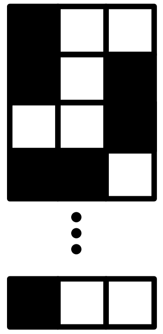
Since only 1/3 of the matrix needs to be covered, the lower bounds for both max-reducer-input and max-reducer-output will be less than for the cross-product that had to cover the entire matrix.

Identifying Candidate Cells

- Given a join condition, the algorithm has to identify an appropriate partitioning of the join attribute domain so that as much of the join matrix as possible can be eliminated, leaving only a small fraction of candidate cells to be covered.
- For simple equality and inequality conditions on a single join attribute, range partitioning based on quantiles tends to work very well. (Finding of approximate quantiles was discussed in a previous module.)
  - This ensures that each partition of S and T tuples has a similar amount of data. Hence eliminating a combination of such partitions significantly reduces input duplication.
- For more complex conditions, in particular those on multiple attributes, finding good partitions of S and T can be challenging.
- Once the blocks of candidate cells have been identified, a cover using r regions has to be found. Due to the possibly irregular shape of candidate blocks, finding a good cover is much more challenging than for the cross-product.

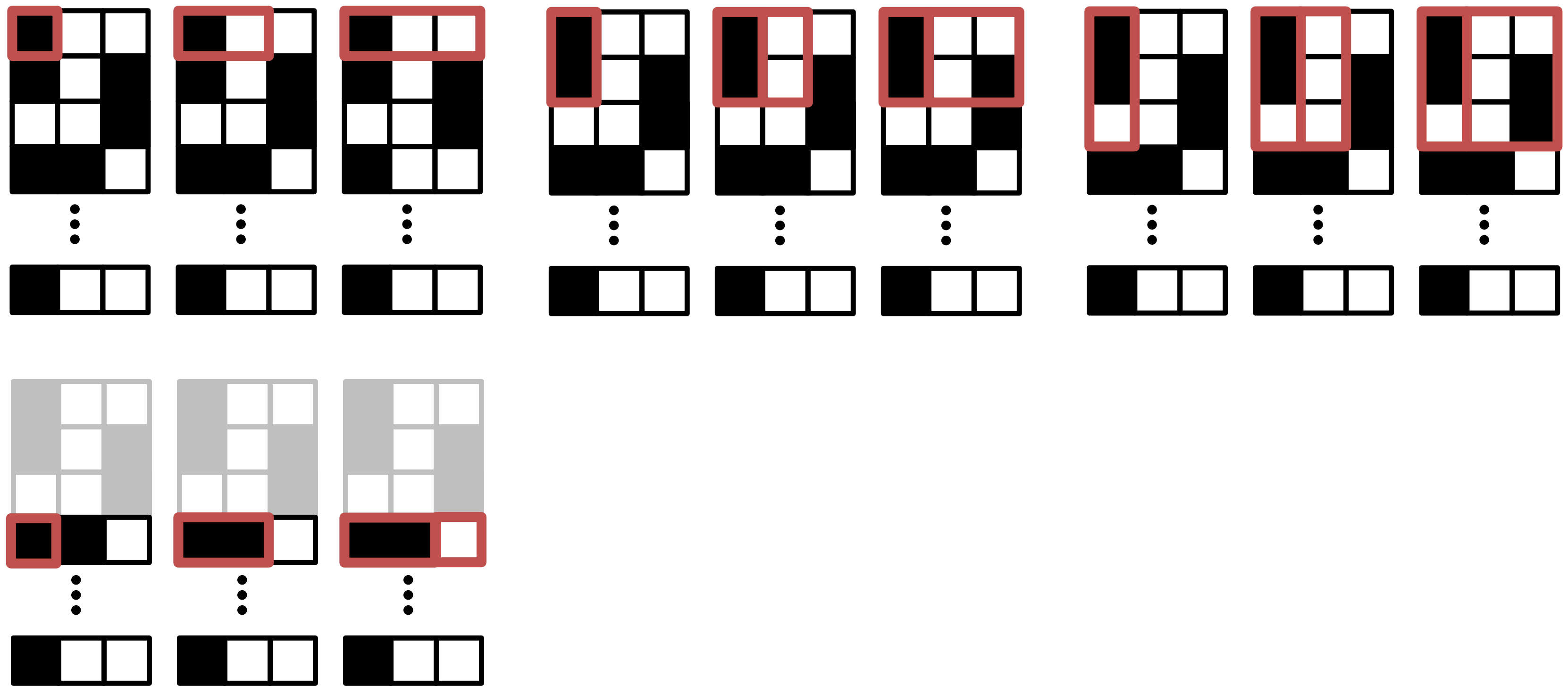
M-Bucket-I

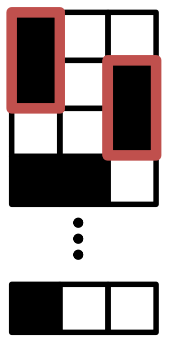
- **M**-Bucket-**I** is a heuristic for finding a cover of all blocks of candidate cells that were identified based on a partitioning of S and T into **m**ultiple buckets. Its goal is to minimize max-reducer-**i**nput.
- M-Bucket-I greedily identifies the best possible cover for a block of rows starting in row 0. It performs a *binary search* on possible max-reducer-input limits.
  - Given a limit, it finds the row block with the highest ratio of candidate-cells-covered to regions-used, respecting the max-reducer-input limit used for the cover. It then continues with the next block of rows not yet covered.
  - If all candidate cells in the matrix can be covered with r regions, M-Bucket-I tries a smaller limit on max-reducer-input. Otherwise it tries a larger limit.
- Once a cover of all candidate cells is found, an algorithm similar to Basic-Theta can be used to implement the join. The main difference is that instead of picking *any* random row for an S-tuple s, it has to pick a random row from within the S-partition that s falls into. Similarly, randomization for T-tuples is limited to the partition the T-tuple belongs to.
- Let us look at an example to better understand this idea.



Join matrix showing all candidate cells shaded. Assume the max-reducer-input limit is set to 3. This means that each region can only have a total of up to three different rows and columns with candidate cells assigned to it.

M-Bucket-I first attempts to cover a block of rows starting with row 0. Each sequence of three steps shows how it explores covering 1, 2, and then 3 rows. Whenever the max-reducer-input limit is reached, a region is "closed" and a new one starts. The ratios of candidate cells covered to regions used are 1, 3/2, and 4/2 for the blocks of 1, 2, and 3 rows, respectively. Hence the 3-row cover wins and M-Bucket-I continues exploring covers starting in row 3.





Winning cover for starting row 0.

M-Bucket-O

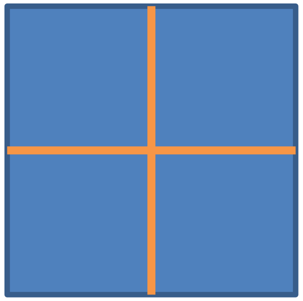
- The M-Bucket-O algorithm is similar to M-Bucket-I, but tries to minimize max-reducer-**o**utput. It performs a binary search over limits on max-reducer-output.
- The main additional challenge for this algorithm is that it has to estimate the number of “true”-valued cells in a region of candidate cells. (Recall that not all candidate cells need to correspond to join result tuples.) This is a challenging problem known as *selectivity estimation for joins* in relational databases. Selectivity estimates can be poor, even for comparably fine-grained histograms of the join attribute’s frequency distribution.

Extension: Memory-Awareness

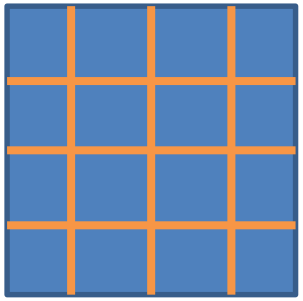
- The cost model we relied on so far was fairly simplistic in that it accounted for computation costs only in terms of the amount of input and output per task. Big-data problems often benefit from a more fine-grained analysis that distinguishes between cases where data fits in memory and where it does not.
  - If data does not fit in memory, it has to be managed and accessed in slower (meaning higher latency and lower bandwidth) external storage such as flash drives or hard disks. For some algorithms this has only a negligible effect on performance. In particular, if an algorithm simply scans through a data set from start to end, then buffering can hide access latency very well. On the other hand, an algorithm that repeatedly jumps to "random" locations in a large file will pay a high performance penalty due to latency.
- All theta-join algorithms introduced in this module can be made memory-aware. This is facilitated by the matrix-based model, which makes it easy to determine how many input tuples from S and T a task will receive.
- Memory-awareness through I/O-optimized local computation:
  - If a Reduce call's input list does not fit in memory, then the local join implementation executed in the Reduce function should not attempt to load all input at once. Instead, a local join implementation optimized for external memory should be used—designing and implementing this is non-trivial.

Extension: Memory-Awareness (Cont.)

- Memory-awareness through region size:
  - To avoid having to deal with accesses to slower storage, we can create smaller regions by choosing a larger number of region keys, r.
  - The best value for r is found through binary search: If the matrix cover for some value of r creates tasks that exceed memory size, then a larger r is explored; and vice versa.
  - Increasing the number of regions will decrease per-task input and output size, but it tends to increase total cost because more duplicates of input tuples have to be created for the additional regions. (See the example below.)
  - Recall that M-Bucket-I explores different limits on max-reducer-input anyway. Hence for M-Bucket-I it suffices to immediately set that limit to the size of Reducer memory.



This cover results in a doubling of input size in the Mappers. Each row intersects with two regions, requiring two copies of the corresponding S-tuple; similarly for T-tuples. Each Reduce call deals with half of S and T.



This cover results in a quadrupling of input size in the Mappers. Each row intersects with four regions, requiring four copies of the corresponding S-tuple; similarly for T-tuples. Each Reduce call only deals with a quarter of S and T.

Experiments: Basic Setup

- All experiments were executed on a cluster consisting of 10 machines with the following specs: quad-core Xeon 2.4GHz CPU, 8 MB cache, 8 GB RAM, and two 250 GB 7.2K RPM hard disks.
- The cluster was running Hadoop 0.20.2 with one machine dedicated as the head node, and the other nine as worker nodes. The default Hadoop configuration was used, setting one Map and Reduce slot per core and HDFS block size of 64MB. Data is stored on all 10 machines.
- For all experiments, r was set to 36. This corresponds to the number of Reduce slots in the cluster, allowing a job to finish in one wave by assigning exactly one key to each Reduce task.
  - For the memory-aware version, r was set to the smallest multiple of 36 for which the input could fit in memory. Then each of the 36 Reduce tasks received the same number of keys.

Datasets

- Both real and synthetic data were explored. The real data set **Cloud** consists of 382 million records, each with 28 attributes, for 28.8 GB total size. These records contain data from cloud reports by ships and land stations.
- Real data sets **Cloud-5-1** and **Cloud-5-2** are independently drawn uniform random samples from Cloud, each containing 5 million records.
- **Synth-α** defines a family of synthetic data sets. For some value of α, Synth-α is a pair of data sets, each containing 5 million records. The records are integers between 1 and 1000. The first data set is generated using the uniform distribution. The second is drawn from a Zipf distribution with parameter α.
  – For $\alpha = 0$ the distribution is perfectly uniform. The larger α, the more skewed the distribution. Distributions with $\alpha > 2$ are considered extremely skewed.

Skew Resistance: Equi-Join

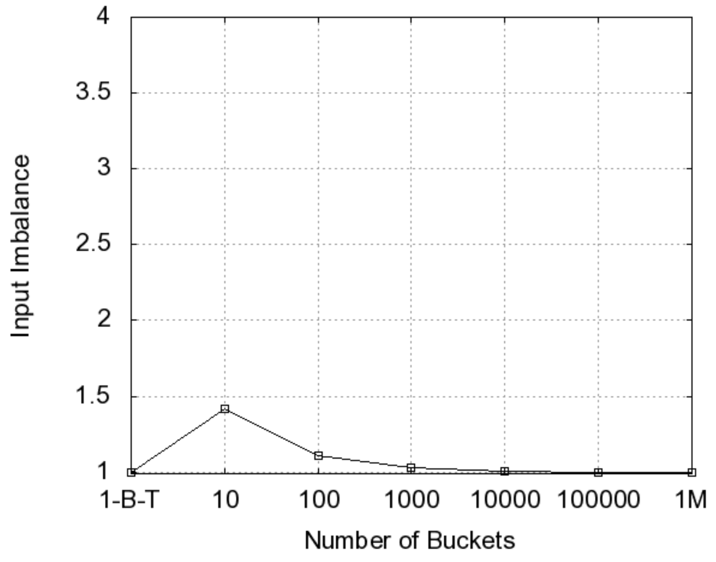
- This experiment compares 1-Bucket-Random to hash+shuffle (Reduce-side join) for an equi-join. The problem is output-size dominated as the 10 million input tuples produce about 25 billion output tuples.
  - Output imbalance measures the ratio between max-reducer-output and average reducer output.
- The results show clearly that as data skew increases, Reduce-side join suffers from unbalanced load distribution. This imbalance is highly correlated with the running time of the job. Contrast this to 1-Bucket-Random, which always balances load very well and retains the fast execution time even for skewed data.

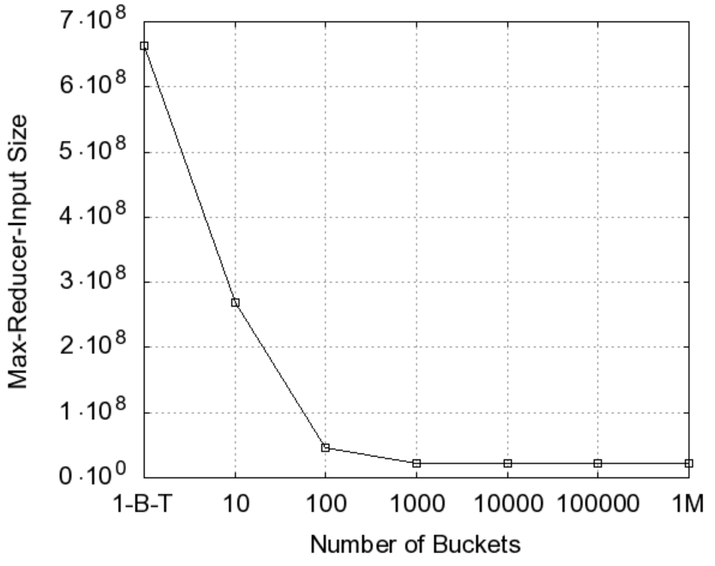
		1-Bucket-Random		Reduce-side join	
Data Set	Output size (billion)	Output imbalance	Runtime (secs)	Output Imbalance	Runtime (secs)
Synth-0	25.00	1.0030	657	1.001	701
Synth-0.4	24.99	1.0023	650	1.254	722
Synth-0.6	24.98	1.0033	676	1.778	923
Synth-0.8	24.95	1.0068	678	3.010	1482
Synth-1	24.91	1.0089	667	5.312	2489

Selective Band-Join

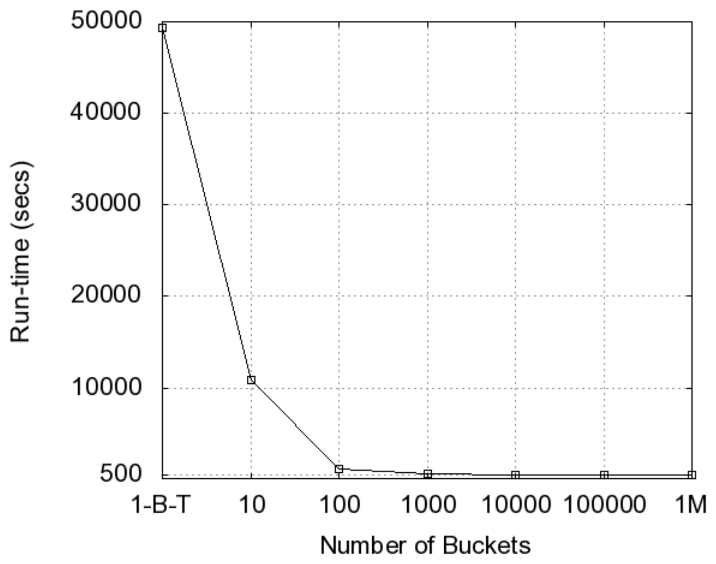
```
SELECT S.date, S.longitude, S.latitude, T.latitude
FROM Cloud AS S, Cloud AS T
WHERE S.date = T.date
  AND S.longitude = T.longitude AND
  ABS(S.latitude - T.latitude) <= 10
```

- This query finds cloud reports made on the same day in neighboring geographical locations.
- The join is input-size dominated, because the 764 million input records (the Cloud data set is used twice for this self-join) only produce 390 million output records.
- The experiments explore the performance of M-Bucket-I, which optimizes for the input-size dominated join case, for different granularities of join matrix partitioning.





The first graph reports input imbalance, measured as max-reducer-input divided by average reducer input. The x-axis shows the granularity of the matrix partitioning used for eliminating regions without join results. (1-B-T uses a single partition, like 1-Bucket-Random.) In all cases, input load is well-balanced. The second graph highlights the importance of using fine-grained matrix partitioning for eliminating regions without join results: 1-B-T covers the entre matrix and hence suffers from excessive input duplication.



This graph shows the corresponding running time of the MapReduce job executing the join. (All numbers are 10-run averages; standard deviation was below 15%.) It is clearly visible how closely running time is correlated with max-reducer-input. This confirms that for input-size dominated joins, it is very important to avoid covering join-matrix regions that cannot produce results.

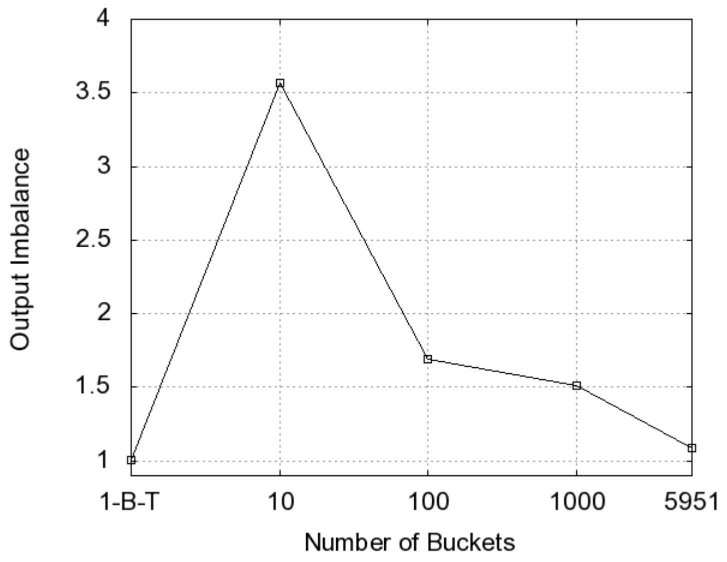
M-Bucket-I Details

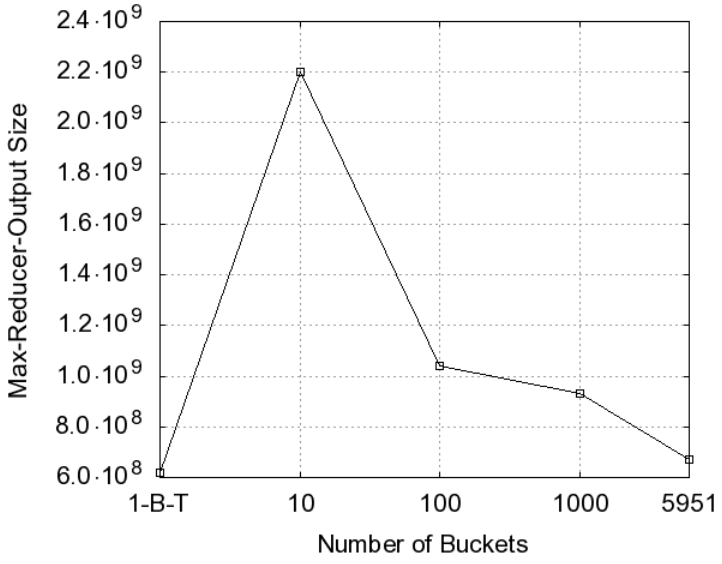
- M-Bucket-I, with a single bucket per input, results in an algorithm that covers the entire matrix, like 1-Bucket-Random. In contrast to 1-Bucket-Random, it uses the more sophisticated heuristic discussed earlier for finding a cover.
- For all experiments, the memory-aware version of M-Bucket-I was used. This made every Reduce call execute in memory, at the possible cost of an overall greater input duplication.
- Total input duplication rate is measured as total Mapper output size divided by total Mapper input size. Input duplication rates were 31.22, 8.92, 1.93, 1.043, 1.00048, and 1.00025 for the experiments with 1, 10, 100, 1000, 10K, 100K, and 1M buckets, respectively.
  - More buckets result in a more fine-grained discovery of candidate cells, reducing the number of cells to be covered.

Not-So-Selective Band-Join

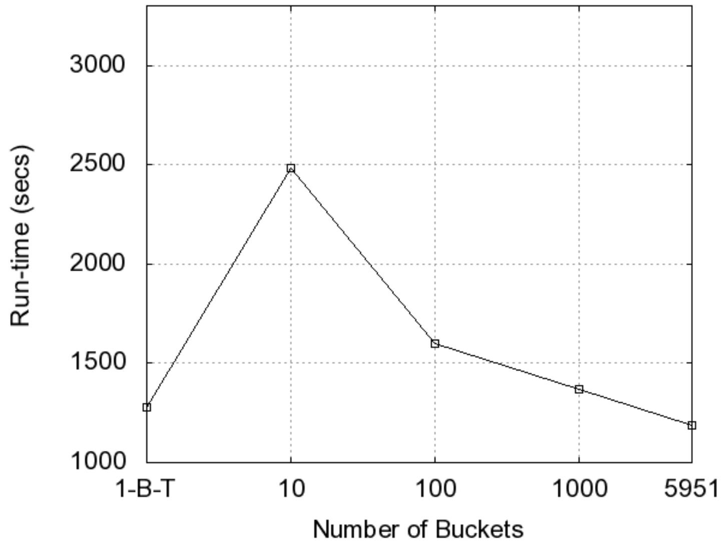
```
SELECT S.latitude, T.latitude
FROM Cloud-5-1 AS S, Cloud-5-2 AS T
WHERE ABS(S.latitude-T.latitude) <= 2
```

- This query finds cloud reports made at similar latitudes.
- The join is output-size dominated, because the 10 million input records (the Cloud data samples were used to avoid excessive running times) produce 22 billion output records.
- The experiments explore the performance of M-Bucket-O, which optimizes for the output-size dominated join case, for different granularities of matrix partitioning.





The first graph reports output imbalance, measured as max-reducer-output divided by average reducer output. The x-axis shows the granularity of the matrix partitioning used for eliminating regions without join results. (1-B-T uses a single partition, like 1-Bucket-Random.) While output is well-balanced for 1-B-T due to randomization, imbalance is fairly high for the coarser partitionings. This is caused by the difficulty of estimating output size for a matrix partition. Even for the most fine-grained partitioning possible (there are only 5951 distinct latitude values in the data), imbalance is still higher than for the simple randomized approach. Since M-Bucket-O does not duplicate output records, the second graph showing max-reducer-output is identical in shape to the first.



This graph shows the corresponding running time of the MapReduce job executing the join. (All numbers are 10-run averages; standard deviation was below 4%.) It is clearly visible how closely running time is correlated with max-reducer-output. Notice that a smaller bucket number still causes more input duplication. However, since the join is output-size dominated, this does not significantly affect the running time. In particular, 1-B-T, which covers the entire join matrix, performs about as well as the most fine-grained partitioning that needs to cover only a small fraction of the join matrix.

M-Bucket-O Details

- M-Bucket-O, with a single bucket per input, results in an algorithm that covers the entire matrix, like 1-Bucket-Random. In contrast to 1-Bucket-Random, it uses a version of the more sophisticated heuristic discussed earlier for M-Bucket-I.
- For all experiments, the memory-aware version of M-Bucket-O was used. This made every Reduce call execute in memory, at the possible cost of an overall greater input duplication.
- Total input duplication rate is measured as total Mapper output size divided by total Mapper input size. Input duplication rates were 7.50, 4.14, 1.46, 1.053, and 1.035 for the experiments with 1, 10, 100, 1000, and 5951 buckets, respectively.

Cost of Collecting Statistics

- The 1-Bucket-Random algorithm and the M-Bucket algorithms for M=1 cover the entire join matrix. Hence they do not need to collect detailed statistics about the join attribute distribution in S and T.
- On the other hand, to choose “good” matrix partitions for eliminating regions without join results, the M-Bucket algorithms for M>1 use quantiles. Hence they need to perform a pre-processing step for finding approximate quantiles.
- The M-Bucket algorithms also incur additional cost for finding a good cover of the regions with candidate cells.
- Comparing these costs provides more insights into the properties of the M-Bucket algorithms, as shown next.

M-Bucket-I on Cloud data set (input-size dominated join):

Step	Number of histogram buckets						
	1	10	100	1000	10,000	100,000	1,000,000
Quantiles	0	115	120	117	122	124	122
Histogram	0	140	145	147	157	167	604
Heuristic	74	9	0.8	1.5	17	118	111
Join	49,384	10,905	1157	595	548	540	536
Total	49,458	11169	1423	861	844	949	1373

M-Bucket-O on Cloud-5 data sets (output-size dominated join):

Step	Number of histogram buckets				
	1	10	100	1000	5951
Quantiles	0	4.5	4.5	4.8	4.9
Histogram	0	26.2	25.8	25.6	25.6
Heuristic	0.04	0.04	0.05	0.24	0.81
Join	1279	2483	1597	1369	1188
Total	1279	2514	1627	1399	1219

Result Discussion

- The tables above show the detailed cost breakdown for the same join queries on the Cloud data discussed before.
- For the input-size dominated join, it is clearly visible that the computation time invested into finding quantiles and the corresponding histogram-style partitioning of the join matrix pays a great dividend in savings in the join computation phase. The savings originate from identifying large regions of the join matrix that do not need to be covered, dramatically reducing input duplication.
- The measurements also show that at some point the benefit of a more fine-grained analysis of the join matrix are outweighed by the higher pre-processing cost.
- For the output-size dominated join, the case with a single bucket is practically tied for first place with the case where the most fine-grained bucketization possible of the matrix was applied.

Extensions

- The presented algorithms were the first thorough study of distributed theta-joins. This work motivated many follow-up papers by various research groups, including
  – Multi-way theta-joins between more than two relations [Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using MapReduce. *Proc. VLDB Endowment*, pages 1184-1195, 2012]
  – More advanced partitioning algorithms that optimize for the weighted sum of input and output [A. Vitorovic, M. Elseidy and C. Koch. Load balancing and skew resilience for parallel joins. IEEE ICDE, pp. 313-324, 2016]

Next, we explore matrix multiplication, a core operation in linear algebra that is expensive and relatively easy to parallelize. It is an important building block in many applications, including machine learning algorithms.

Matrix Multiplication and Regression

- Linear algebra is an important mathematical tool for data analysis. Equations in linear algebra are naturally expressed as manipulations of matrices and vectors.
- Recall the graph analysis discussion from an earlier module. Problems such as finding paths in a graph and computing PageRank can be expressed as matrix multiplication problems. This module introduces parallel matrix multiplication in the context of linear regression, a classic method for statistical analysis.

Linear Regression

- Linear regression is a popular prediction technique. As introduced earlier, prediction is a supervised learning method: Given a training data set with attributes $X_1,\ldots,X_d$, and Y, a model $f:(X_1,\ldots,X_d)\rightarrow Y$ is trained. This function can then be used to predict the unknown output y for a given input record $(x_1,\ldots,x_d)$. For *prediction* problems, Y is a continuous attribute.
- Linear regression is considered a parametric learning technique, because its goal is to learn the best parameter values for an expert-selected function "template."
  - A linear regression function is defined as $f(X_1,\ldots,X_n) = \theta_0 + \theta_1 X_1 + \cdots + \theta_n X_n$, written more compactly as $\mathrm{Y} = \boldsymbol{\theta}^T \boldsymbol{X}$. Output y is a scalar; input **X** and parameter $\boldsymbol{\theta}$ are n-dimensional *vectors*.
- Linear regression is more flexible than it may seem. We can introduce new input variables that are non-trivial functions of the original input attributes, e.g., $X_{n+1} = X_1^2 + X_2^2$ or $X_{n+2} = X_3 X_5$. The training data would be augmented with the corresponding new columns, whose values are trivial to compute from the values in the original columns.

Optimal Solution

- The goal of linear regression is to find the vector $\boldsymbol{\theta}^*$ that minimizes squared error $\sum_{i=1}^{m}(\boldsymbol{\theta}^T\boldsymbol{x}_i - y_i)^2$ over all training records $(\boldsymbol{x}_i, y_i)$.
- It has been shown that the optimal solution can be computed as $\boldsymbol{\theta}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$.
  - Here $\boldsymbol{X}$ is an n-by-d matrix, where the input values of training record $\boldsymbol{x}_i$ make up the i-th row.
  - Similarly, **y** is an n-by-1 vector, corresponding to the Y-column of the training data.
  - An example is shown below.

X_1	X_2	X_3	Y
1	0	5	8
4	2	3	9

Training data

$$\begin{bmatrix} 1 & 0 & 5 \\ 4 & 2 & 3 \end{bmatrix}$$

Matrix ***X***

$$\begin{bmatrix} 1 & 4 \\ 0 & 2 \\ 5 & 3 \end{bmatrix}$$

Transposed matrix $\boldsymbol{X}^T$

$$\begin{bmatrix} 8 \\ 9 \end{bmatrix}$$

Vector **y**

Matrix Multiplication

- The formula for the computation of the optimal parameter vector $\boldsymbol{\theta}^*$ requires three matrix products (one of them the multiplication with vector $\mathbf{y}$), and a matrix inversion.
- We will discuss distributed matrix multiplication, but first review the basics:
  - A u-by-v matrix has u rows and v columns. Multiplying an a-by-b with a b-by-c matrix will create an a-by-c matrix.
  - The entry in row i and column j of the result matrix is equal to the dot product of the i-th row vector of the first matrix with the j-th column vector of the second matrix.
- The example below illustrates the matrix product.

$$\boxed{\begin{matrix}1 & 0 & 5\\ 4 & 2 & 3\end{matrix}} \quad \boxed{\begin{matrix}0 & 1\\ 2 & 3\\ 4 & 5\end{matrix}} \quad = \quad \boxed{\begin{matrix}1*0+0*2+5*4 = 20 & 1*1+0*3+5*5 = 26\\ 4*0+2*2+3*4 = 16 & 4*1+2*3+3*5 = 25\end{matrix}}$$

Parallel Matrix Multiplication: Row-by-Column

- The regular structure of matrices makes it comparably easy to identify opportunities parallelization. Recall that each cell of the result matrix is the dot product of a row in the first with a column in the second input matrix. Since different rows in the first and different columns in the second matrix are processed independently, this suggests a row-wise partitioning for the former and a column-wise partitioning of the latter. In general,

$$\boldsymbol{AB} = \begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \vdots \\ \boldsymbol{A_r} \end{bmatrix} [\boldsymbol{B_1} \quad \boldsymbol{B_2} \quad \cdots \quad \boldsymbol{B_c}] = \begin{bmatrix} \boldsymbol{A_1} \times \boldsymbol{B_1} & \boldsymbol{A_1} \times \boldsymbol{B_2} & \cdots & \boldsymbol{A_1} \times \boldsymbol{B_c} \\ \boldsymbol{A_2} \times \boldsymbol{B_1} & \boldsymbol{A_2} \times \boldsymbol{B_2} & \cdots & \boldsymbol{A_2} \times \boldsymbol{B_c} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{A_r} \times \boldsymbol{B_1} & \boldsymbol{A_r} \times \boldsymbol{B_2} & \cdots & \boldsymbol{A_r} \times \boldsymbol{B_c} \end{bmatrix}$$

- Here each $\boldsymbol{A}_i$ and $\boldsymbol{B}_j$ is a matrix, containing some of $\boldsymbol{A}$'s rows and $\boldsymbol{B}$'s columns, respectively. Each product $\boldsymbol{A}_i\boldsymbol{B}_j$ can be computed independently.
- Notice the relationship to the relational cross-product, because every $\boldsymbol{A}_i$ has to be multiplied with every $\boldsymbol{B}_j$. Hence we can use 1-Bucket-Random to compute the final result.
  - For best results, let each $\boldsymbol{A}_i$ and $\boldsymbol{B}_j$ correspond to a *single* row or column, respectively. This way 1-Bucket-Random works with the finest problem granularity, allowing it to better balance load.

Example for Row-by-Column Partitioning

Consider ***AB*** example $\boxed{\begin{array}{ccc}1&0&5\\4&2&3\end{array}}\ \boxed{\begin{array}{cc}0&1\\2&3\\4&5\end{array}} = \boxed{\begin{array}{cc}20&26\\16&25\end{array}}$

Leave ***B*** as is, but partition ***A*** into $\boldsymbol{A}_1 = \boxed{\begin{array}{ccc}1&0&5\end{array}}$ and $\boldsymbol{A}_2 = \boxed{\begin{array}{ccc}4&2&3\end{array}}$.

Terms $\boldsymbol{A}_1\boldsymbol{B}$ and $\boldsymbol{A}_2\boldsymbol{B}$ produce result matrix rows $\boxed{\begin{array}{cc}20&26\end{array}}$ and $\boxed{\begin{array}{cc}16&25\end{array}}$:

$$\begin{tikzcd}
 & \boxed{\begin{array}{ccc}1&0&5\end{array}}\ \boxed{\begin{array}{cc}0&1\\2&3\\4&5\end{array}} = \boxed{\begin{array}{cc}20&26\end{array}} \arrow[dr] & \\
\boxed{\begin{array}{ccc}1&0&5\\4&2&3\end{array}} \arrow[ur] \arrow[dr] & & \boxed{\begin{array}{cc}20&26\\16&25\end{array}} \\
 & \boxed{\begin{array}{ccc}4&2&3\end{array}}\ \boxed{\begin{array}{cc}0&1\\2&3\\4&5\end{array}} = \boxed{\begin{array}{cc}16&25\end{array}} \arrow[ur] &
\end{tikzcd}$$

Parallel Matrix Multiplication: Column-by-Row

- Somewhat less obvious than the row-by-column approach, matrix multiplication can also be parallelized by partitioning the first matrix by column, and the second by row!
- To see why this is possible, note that the value in row i and column j of the result matrix is equal to ***A***[i,0]***B***[0,j]+***A***[i,1]***B***[1,j]+…+***A***[i,r_a]***B***[c_b,j], where r_a and c_b are the number of ***A***'s rows and ***B***'s columns, respectively. (These numbers have to be identical for the matrix product to be defined.) Hence the product of column vector ***A***[*,k] and row vector ***B***[k,*] produces the individual terms of type A[*,k]B[k,*] needed for the summation.
- This idea is best understood through an example.

Example for Column-by-Row Partitioning

Consider ***AB*** example $\begin{bmatrix} 1 & 0 & 5 \\ 4 & 2 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 1*0+0*2+5*4 = 20 & 1*1+0*3+5*5 = 26 \\ 4*0+2*2+3*4 = 16 & 4*1+2*3+3*5 = 25 \end{bmatrix}$

There are three products of ***A***'s column vectors with the corresponding row vectors of ***B***:

$$\begin{bmatrix} 1 \\ 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 1*0 & 1*1 \\ 4*0 & 4*1 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 2 \end{bmatrix} \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 0*2 & 0*3 \\ 2*2 & 2*3 \end{bmatrix} \qquad \begin{bmatrix} 5 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 5 \end{bmatrix} = \begin{bmatrix} 5*4 & 5*5 \\ 3*4 & 3*5 \end{bmatrix}$$

Matrix sum $\oplus$

$$\begin{bmatrix} 1*0+0*2+5*4 = 20 & 1*1+0*3+5*5 = 26 \\ 4*0+2*2+3*4 = 16 & 4*1+2*3+3*5 = 25 \end{bmatrix}$$

MapReduce Algorithm for Column-by-Row

- The algorithm first performs a distributed equi-join of any A[i,x] with any B[y,j], using condition x=y. The corresponding product is emitted with key (i,j).
- Another MapReduce job then processes the join output by grouping by key and adding all values in each group.
- The program below only shows the join phase. The second job is identical to Word Count per result-matrix-cell index.

```
map( matrixID, row, col, val ) {
 // Partition A into columns
 if (matrixID = A)
   emit( col, (matrixID, row, val) )
 else  // Partition B into rows
   emit( row, (matrixID, col, val) )
}
```

```
// Reduce receives entries A[i,k] and B[k,j] for different i and j. It emits
// all products A[i,k]*B[k,j] with key (i,j), because this is the
// contribution for result cell [i,j].
reduce( common_A_col_B_row, [(matrixID, index, val),…] ) {
 for each (matrixID, index, val) do
   if (matrixID = A) then A_list.add( index, val )
   else B_list.add( index, val )

 for each Aik in A_list
   for each Bkj in B_list
     emit( (Aik.index, Bkj.index), Aik.val * Bkj.val )
}
```

Comparison of Row-by-Column versus Column-by-Row

- The main cost of both algorithms is related to the data transfer. It strongly depends on the data properties, in particular the sparseness of the input matrices and distribution of non-zero values over their cells.
  - Note that both algorithms work correctly for sparse matrix representation when only cells with non-zero value are stored.
- The row-by-column approach uses 1-Bucket-Random, which duplicates the different matrix blocks in Map. Assuming p Reduce calls with $\sqrt{p}$ partitions of ***A*** and ***B*** each, both ***A*** and ***B*** would be sent $\sqrt{p}$ times in total from Mappers to Reducers.
- The column-by-row approach does not duplicate data in the Map phase. However, its Reducers write out an intermediate result of individual contributions to cells in the output matrix. The size of this intermediate result depends on the number of A[i,k] and B[k,j] values in the input list of each Reduce call for key (i,j). This result is also read, then transferred from Mappers to Reducers in the post-processing job.

Multiplying a Matrix with its Transpose

- When multiplying a matrix with its transpose, the column-by-row approach can be optimized further. Notice that by definition the k-th column in matrix ***A*** is identical to the k-th row of its transpose.
- If matrix ***A*** is stored column-wise, all $A[*,k]A^T[k,*]$ can already be computed in the Mappers, letting the Reducers perform the final aggregation step. *This eliminates the additional post-processing phase.*
  - For linear regression, this applies to $\boldsymbol{X}^T\boldsymbol{X}$. (Since ***X*** is stored row-wise, $\boldsymbol{X}^T$ is stored column-wise.) Term $\boldsymbol{X}^T\boldsymbol{y}$ can be computed similarly, by emitting all $X^T[*,k]y[k] = X[k,*]y[k]$, which are all stored in row k of the input matrix, in the same Map call.
- The program for computing the product of a matrix (that is stored column-wise) and its transpose is shown below.

```
// There is only one input matrix. Map reads
// an entire column of it
map( col, [(row, val), (row, val),…] ) {
 for each (r1, v1) in valueList
   for each (r2, v2) in valueList
     emit( (r1, r2), v1 * v2 )
}
```

```
// Reduce receives all A[i,k]*B[k,j] for result cell [i,j] and sums them up
reduce( (i,j), [val, val,…] ) {
 sum = 0
 for each val in input list do
   sum += val

 emit( (i, j), sum )
}
```

Matrix Product in Spark

- Spark offers linear algebra operations such as matrix product in the MLlib linalg package.
  - Both dense and sparse matrix representation are supported.
- Take a look at the source code to find out more about the underlying implementation. Most likely it uses block partitioning.
- In practice, it is not easy to tune block size, even for dense matrix operations. We explored this in a research paper.
  - Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. Abstract Cost Models for Distributed Data-Intensive Computations. In *Distributed and Parallel Databases*, Springer. 2018 (accepted for publication)

Matrix Multiplication in Machine Learning

- Many other machine learning techniques can be implemented using matrix products, including:
  - Locally Weighted Linear Regression
  - Naïve Bayes
  - Gaussian Discriminative Analysis
  - K-means clustering
  - Logistic Regression
  - Neural Network (for backpropagation)
  - Principal Component Analysis
  - Independent Component Analysis
  - Expectation Maximization (EM) with Mixture of Gaussian as underlying model
  - Support Vector Machine (SVM) with linear kernel
- For further discussion consult [Chu, Kim, Lin, Yu, Bradski, Ng, and Olukotun. Map-Reduce for Machine Learning on Multicore. In Proc. of *Advances in Neural Information Processing Systems (NIPS)*, 2006]

Summary

- When dealing with big data in a distributed system, arguably the most important decision is how to partition the data. Partitioning should achieve two goals:
  - Each task receives a small subset of the data.
  - Each task can be performed independently of the others, possibly requiring a *small* amount of data to be exchanged.
- Modeling data partitioning as a matrix or array covering problem simplifies algorithm design and enables analysis of algorithm properties.
- Randomization plays a key role in transforming a matrix or array cover into a parallel algorithm. It can also simplify the process of proving properties, in particular lower and upper bounds of costs or performance metrics.
- The properties of a matrix or array cover depend heavily on the given problem. For example, sometimes region boundaries indicate data replication (theta-join), sometimes they do not (frequency computation).

Summary (Cont.)

- The relational equi-join and cross-product pattern also appeared in ensemble predictions and matrix product, highlighting the general importance of joins.
- The matrix-multiplication approaches presented in this module support both dense and sparse matrix representations (which store only non-zero cells). The choice depends on sparseness and distribution of non-zero values over matrix cells. The common problem of multiplying a matrix with its own transpose admits a more efficient distributed algorithm for column-by-row partitioning.

5. CYK: Q1

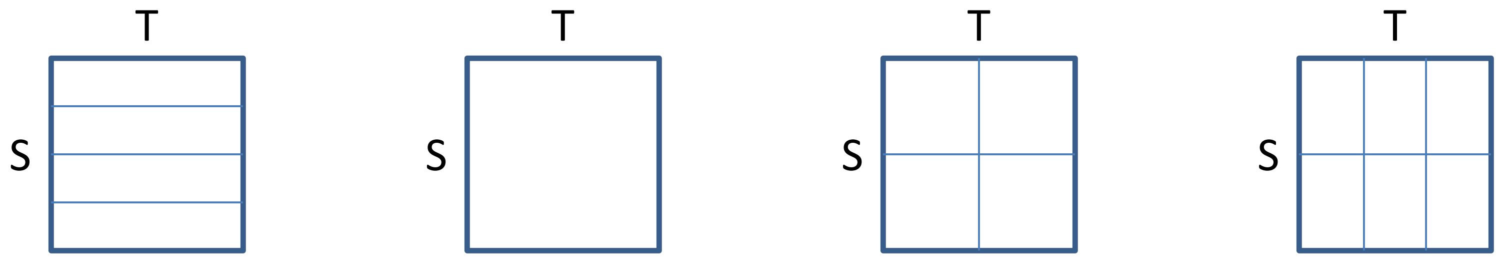
- Given a large set of (species, color) records, the goal is to compute the total number of occurrences for each species-color pair. A programmer considers three MapReduce programs:
  - P1: For input record (S, C), Map emits (S, C) as key and 1 as value. P1 uses the default hash Partitioner. Reduce scans through the input list once to compute the desired count(s).
  - P2: For input record (S, C), Map emits (S, C) as key and 1 as value. P2 uses a custom Partitioner that hashes on the S value only. Reduce scans through the input list once to compute the desired count(s).
  - P3: For input record (S, C), Map emits S as key and C as value. P3 uses the default hash Partitioner. Reduce scans through the input list once to compute the desired count(s).
- For each of the following scenarios, choose which of the three programs would be a good choice or a potentially risky choice:
  1. Heap memory is very limited and the programmer is worried about out-of-memory exceptions.
  2. The programmer wants to use a large number of machines and is trying to create work for all of them.
  3. The programmer knows that the data is very skewed, in particular if some species are observed much more frequently than others, and s/he is worried about balancing work across different machines.

CYK: Q2

- For each statement about the use of randomization for assigning data records to Reduce tasks during the theta-join computation, select if it is true or false:
  1. Randomization is very useful for balancing load across Reducers.
  2. Randomization is very useful for lowering the total cost of the Reduce phase.
  3. Use of randomization simplifies the program.
  4. Randomization makes it more difficult to analyze the properties of the algorithm.

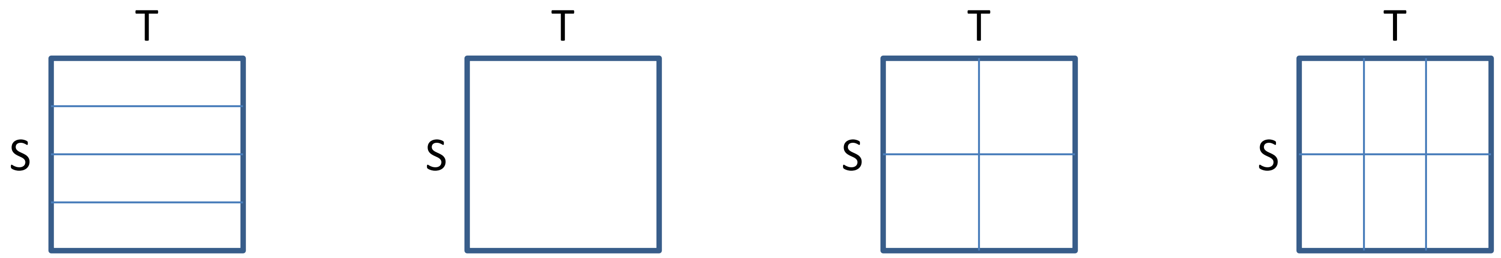
CYK 3

- Consider the theta-join computation problem using 1-Bucket-Random. For each join-matrix cover, state how many copies of each S- and T-tuple are transferred from Mappers to Reducers. (Note: For a Map function that emits an input record exactly once, the answer would be "1 copy.")



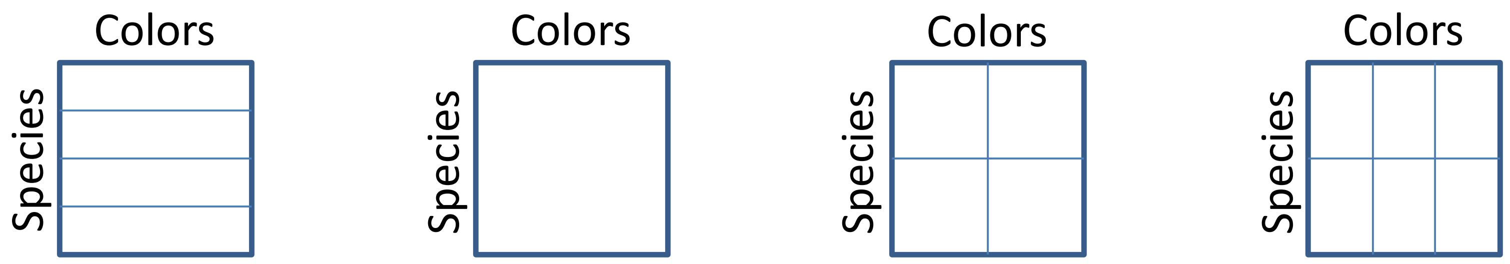
CYK 4

- Consider the theta-join computation problem using M-Bucket-I. For each join-matrix cover, state how many copies of each S- and T-tuple are transferred from Mappers to Reducers. (Note: For a Map function that emits an input record exactly once, the answer would be “1 copy.”)



CYK 5

- Consider the frequency-counting problem for species and colors. (Determine the total number of occurrences of each species-color combination.) For each matrix cover, state how many copies of each S- and T-tuple are transferred from Mappers to Reducers. (Note: For a Map function that emits an input record exactly once, the answer would be "1 copy.")



CYK 6

- Consider the row-by-column approach to distributed matrix multiplication. For each statement, select if it is true or false, assuming we only consider “reasonable” MapReduce implementations that effectively use many machines.

1. This approach can always be implemented as a single Map-Reduce job.
2. For multiplying a matrix with its transpose, it can be implemented as a Map-only job.
3. There is no duplication of input records during the Map phase.
4. The algorithm also works correctly and efficiently for sparse matrices that only store cells with non-zero values.
5. This problem is closely related to join computation.

QCYK 7

- Consider the column-by-row approach to distributed matrix multiplication. For each statement, select if it is true or false, assuming we only consider “reasonable” MapReduce implementations that effectively use many machines.

1. This approach can always be implemented as a single Map-Reduce job.
2. For multiplying a matrix with its transpose, it can be implemented as a Map-only job.
3. There is no duplication of input records during the Map phase.
4. The algorithm also works correctly and efficiently for sparse matrices that only store cells with non-zero values.
5. This problem is closely related to join computation.

References

- A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In Proc. ACM SIGMOD Int. Conf. on Managament of Data, pages 949-960, 2011
  - https://scholar.google.com/scholar?cluster=13110021479290984892&hl=en&as_sdt=0,22
- Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. Abstract Cost Models for Distributed Data-Intensive Computations. In *Distributed and Parallel Databases*, Springer. 2018 (accepted for publication)
  - http://www.ccs.neu.edu/home/mirek/papers/2018-DAPD-AbstractCostModels-preprint.pdf
- Chu, Kim, Lin, Yu, Bradski, Ng, and Olukotun. Map-Reduce for Machine Learning on Multicore. In Proc. of *Advances in Neural Information Processing Systems (NIPS)*, 2006
  - https://scholar.google.com/scholar?cluster=7784975056176979771&hl=en&as_sdt=0,22

References

- Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using MapReduce. *Proc. VLDB Endowment*, pages 1184-1195, 2012
  - https://scholar.google.com/scholar?cluster=8095041193631575005&hl=en&as_sdt=0,22
- A. Vitorovic, M. Elseidy and C. Koch. Load balancing and skew resilience for parallel joins. IEEE ICDE, pp. 313-324, 2016
  - https://scholar.google.com/scholar?cluster=1010206575348448478&hl=en&as_sdt=0,22