Intelligent Partitioning

Diego Rivera Correa

Slides owned by 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

- Scalable distributed data processing requires data to be partitioned and assigned to tasks that can be executed independently. With more tasks, each task should receive less data and finish faster.

- How do we find a good partitioning, analyze its properties, and reason about its performance?

Introduction

- We will discuss a general strategy that starts with the most fine-grained partitioning and then arranges 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 examples.

Let us start with an easy problem: 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 partition scheme, we first need to identify and formalize the space of all possible partitionings 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.

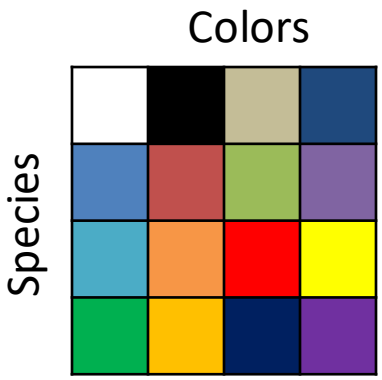
Design Space

- Let |S| and |C| denote the number of different species and colors, respectively.

- Partitioning the given problem requires assigning each of the |S|·|C| possible pairs to some task responsible for counting 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. Each cell, representing the most fine-grained problem partitioning, corresponds to a desired count.




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

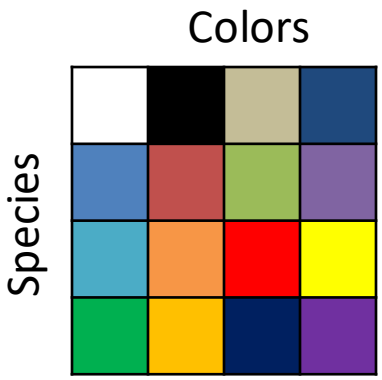
```
reduce( (S, C), [n1, n2,…] ) {
  frequency = 0

  for all n in input list do
    frequency += n

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

Frequency Counting using Pairs

- The Pairs approach assigns a unique key to each cell. These keys can then be assigned randomly to tasks. We show the corresponding algorithm:
  - 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 )
```

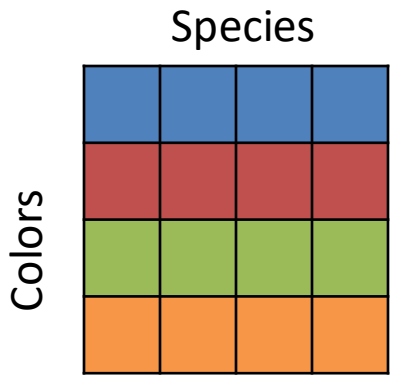
```
reduce( (S, C), [n1, n2,…] ) {
  frequency = 0

  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 this partitioning, the algorithm uses only the species as the key. Since each Reduce call now works with an entire row, a 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**: Can both approaches use Combiners and in-mapper combining?
- **Code complexity**: Which approach is easier to understand?
- **Key Space:** Which approach has more intermediate keys?
- **Memory & Load Balancing:** Which approach uses a data structure? Which approach could scale to more reduce() calls?

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.

Comparison of Pairs versus Stripes

- **Key space**: Pairs manages $O(|S| \cdot |C|)$ intermediate keys, Stripes only $O(|S|)$. This does not affect the overhead for the application master (which depends on the number of tasks) but results in more Reduce function calls for Pairs. On the other hand, each Reduce call for Stripes will be more expensive.

Comparison of Pairs versus Stripes (cont.)

- **Memory**: The Map functions of both approaches do not use any local variables. Pairs' Reduce function requires a single variable, while Stripes' Reduce must 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.

Comparison of Pairs versus Stripes (cont.)

- **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 always emulate Pairs, e.g., 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.

Beyond Pairs and Stripes

- Each matrix cell corresponds to a value of interest—the number of occurrences of a species-color combination—and therefore each matrix cell needs to be computed by some function call in a task.

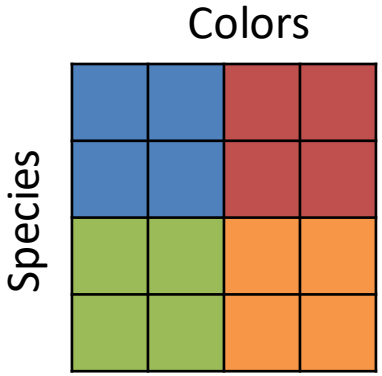
- We can model this by assigning each matrix cell to exactly one of r tasks. Stated differently, the matrix needs to be completely covered by r regions, each corresponding to a different task.

Beyond Pairs and Stripes

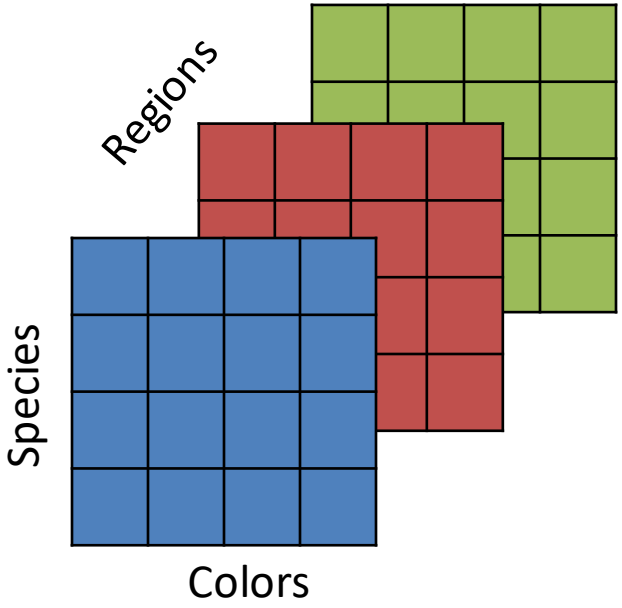
- The best cover should be selected depending on data distribution and computation task.

- This approach generalizes beyond matrices 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 right half could be earthy colors, while the left 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 partitioning.

- Partitioning granularity determines the number of distinct keys and hence Reduce function calls.

Evaluating a Matrix Cover

- 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)...

Evaluating a Matrix Cover

- 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, we 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.

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 design a custom Partitioner to balance load.

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

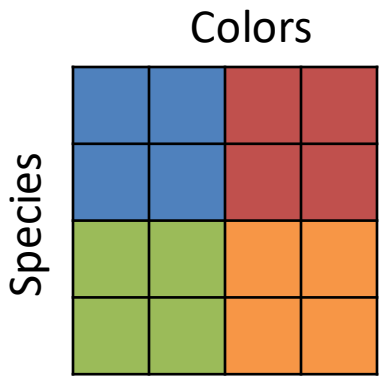
- Instead of the clumsy if-then-else statements, the regions can be encoded more elegantly as a sequence of ranges in each dimension.

- 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.

“Randomized” Block Partitioning

- Finding good ranges to define the blocks requires knowledge about the input data distribution. Consider 2-by-2 blocks, but without pre-defined ranges. Instead, we use a hash function H_S to map a species to “up” vs “down” and another hash function H_C to map a color to “left” vs “right.”

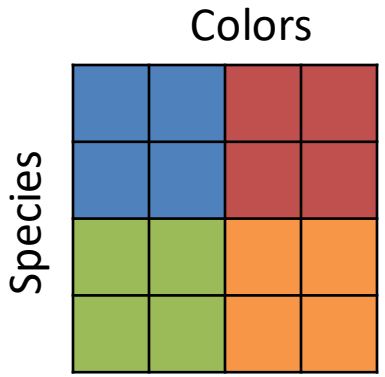
A-by-B block partitioning for A=2, B=2.



“Randomized” Block Partitioning

- In general, consider a partitioning with A rows and B columns. For input (s, c), Map emits value (s, c) with key ($H_S(s)$ mod A, $H_C(c)$ mod B). Then frequency f(s, c) is computed by the corresponding Reduce call.

A-by-B block partitioning for A=2, B=2.



“Randomized” Block Partitioning

- Hash functions assign species and colors “randomly” to row and column blocks but guarantee that the same species will always end up in the same row (similarly for colors and columns).
  - This is important for correctness.

- Note that partitioning here did not introduce data duplication. For problems with different semantics, e.g., theta-joins, this will change.

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

Theta-Joins

- The idea of modeling partitioning 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 affect the analysis of the resulting algorithms.

Problem Definition

- Earlier we discussed distributed equi-joins, 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.

Problem Definition

- Our goal is to partition any given theta-join computation such that job completion time on a given number of worker machines is minimized. (Alternative goals, e.g., maximizing throughput, are not considered here.)

Challenges

- The join techniques discussed so far suffer from severe limitations in the context of theta-joins:
  - Partition+broadcast
  - Hash+shuffle

Challenges

- The join techniques discussed so far suffer from severe limitations in the context of theta-joins:
  - Partition+broadcast can check each S-tuple against all tuples in T (which is broadcast to all tasks) to find the matching pairs (s_i, t_j). While this enables it to implement any theta-join, it will only be efficient if T is small.
  - Hash+shuffle works well when both inputs are big, but it neither generalizes beyond equi-joins nor scales well for join attributes with a small domain or heavily skewed input distribution.

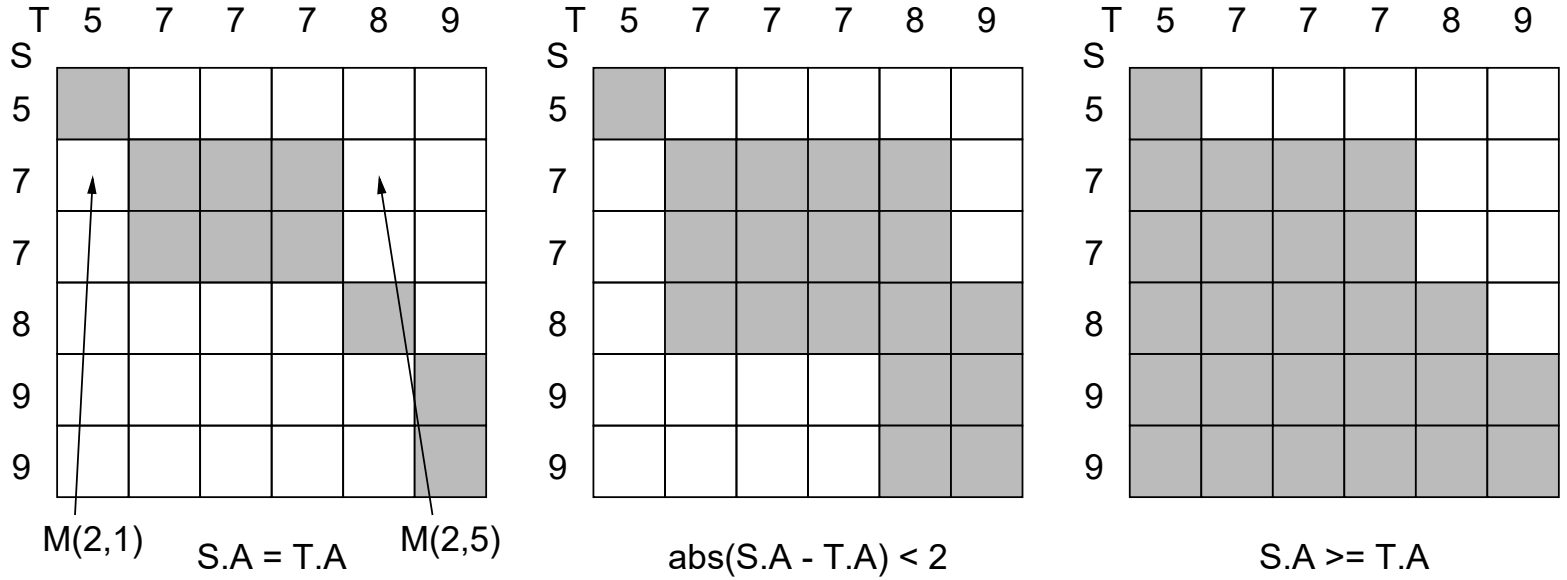
Challenges

- To minimize job completion time, we want a partitioning that minimizes the amount of work assigned to the machine doing the most work. (This machine determines the end of the job!)

- The matrix-cover idea will prove useful for reasoning about this problem and possible solutions.

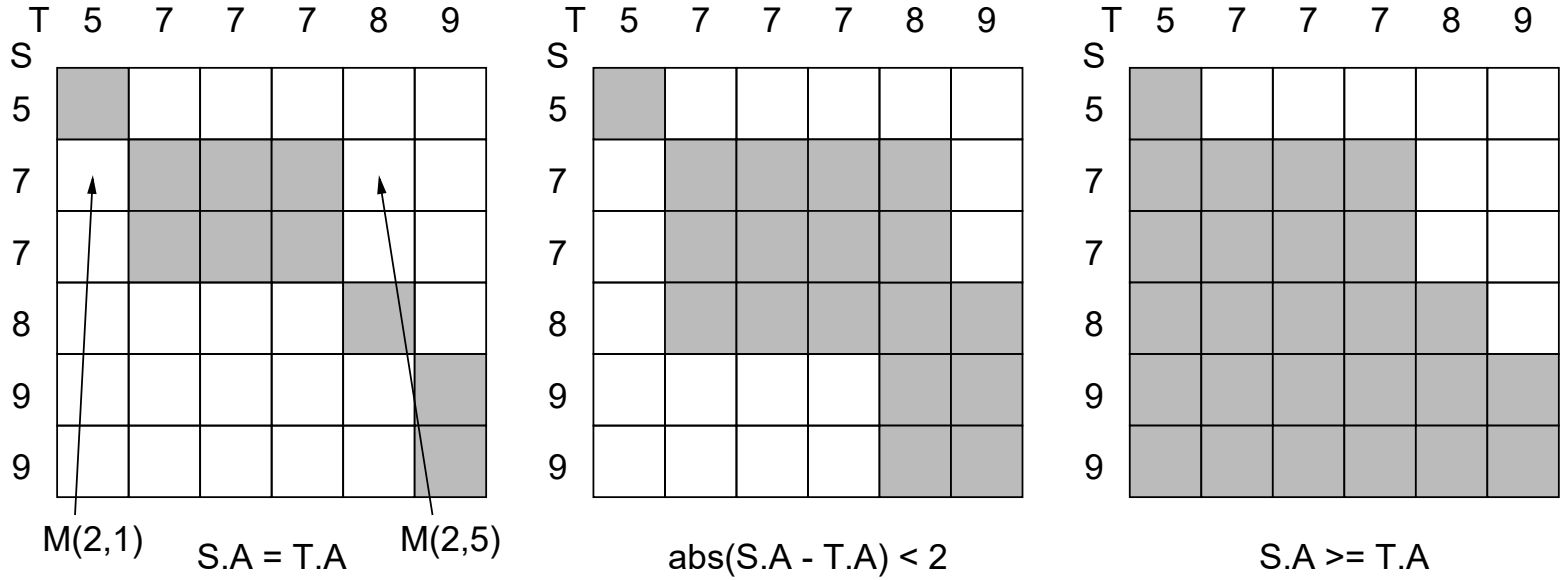
Theta-Join Matrix

- Recall that any theta-join is a subset of the Cartesian 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).



Theta-Join Matrix

- 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 a join result; and "false" otherwise.



Join Matrix in Practice

- The join matrix as discussed so far encodes the exact join result. If it was available from the start, then we would not need to execute the join—the result would be readily available in the matrix.

- Hence a practical join algorithm cannot usethe join matrix. Why do we need it then?

Join Matrix in Practice

- For algorithm analysis: Matrix properties determine algorithm properties and performance, no matter if the algorithm knows the matrix.

Join Matrix in Practice

- If the algorithm needs matrix statistics, e.g., the number of input and output tuples in a region, we can design techniques to estimate them. Here we must ensure that estimation cost is negligible compared to “inherent” join cost.

- E.g., it does not make sense to spend 1 hour on estimating matrix statistics if a simple partition+broadcast implementation computes the join in 10 minutes.

Matrix Cover

- We now explain how a cover of join matrix M defines a distributed theta-join algorithm. The discussion uses MapReduce terminology for simplicity; the same ideas extend to Spark.

- Consider output pair (s_2=7, t_1=5) in cell M(2,1) for join S.A >= T.A and assume it is emitted by the Reduce call for some key K. We say "key K covers M(2,1)" or "the Reduce call for K covers M(2,1)."

Matrix Cover

- For a Reduce call to emit (s_2, t_1), it needs s_2 and t_1. Hence Map must emit these input tuples with key K.

- Putting things together, we observe that every "true"-valued cell in M must be covered by some key, which determines the input tuples sent to the corresponding Reduce call.

- "False"-valued cells need not be covered but covering them does not jeopardize correctness: The Reduce call can verify the join condition and remove input-tuple pairs that violate the join condition.

Covering Candidate Cells

- How does the algorithm know which matrix cells to cover if it does not know the join matrix?

- It “plays it safe” by ensuring that some easy-to-find superset of the “true”-valued cells is covered. We refer to this superset as candidate cells.

Covering Candidate Cells

- What if we declare the *entire join matrix* as candidate cells? This guarantees correctness but may cause performance degradation when many “false”-valued cells are covered.
  - Covering such a cell causes extra duplication for sending the corresponding input tuples to Reducers.
  - It also causes extra computation in Reduce for checking and removing the pair.
- Ideally, only a few “false”-valued join-matrix cells are covered.

Non-Overlapping Cover

- We established that each “true”-valued join-matrix cell must be covered by at least 1 key, but what if it is covered by multiple keys? That causes undesirable output duplicates.

Non-Overlapping Cover

- Consider s_2=7, s_3=7, and t_2=7. Pairs (s_2, t_2) = (7, 7) and (s_3, t_2) = (7, 7) look the same, but are *not duplicates*, because they are produced by two distinct S-tuples.

- On the other hand, emitting (s_2, t_2) more than once is undesirable, because the output would contain additional (7, 7) tuples that should not be there.

Non-Overlapping Cover

- Undesirable output duplicates could be removed in post-processing, but its extra work that needs to be performed.

- For these reasons, we focus on non-overlapping covers, where no matrix cell is covered by more than 1 key.

Matrix-Cover Cost Model

- For a given join problem, there could be many ways to cover (a superset of) all “true”-valued join-matrix cells with non-overlapping regions. How do we select the best?

- We need a cost model that quantifies the relationship between matrix-cover properties and running time.

Matrix-Cover Cost Model

- Even for a given data partitioning, it is difficult to accurately predict running time of a distributed computation.
  - In addition, we must explore the space of all possible partitionings to find the matrix cover with the minimal running time.

- We therefore strive for a cost model that balances simplicity/robustness and accuracy.

Cost-Model Derivation

- We now derive a simple cost model that enables strong analytical results, while also being sufficiently accurate.

- First notice that any MapReduce theta-join implementation must read the input from HDFS into Mappers and write the output back from Reducers to HDFS.
  - Since these costs do not depend on the partitioning strategy, i.e., the matrix cover, they do not help us determine the winner and we therefore ignore them.

Cost-Model Derivation

- Reducers perform the actual join work, while Mappers just duplicate and shuffle the input to the Reduce calls.

- Hence we will focus on the cost of the Reduce phase.

Cost-Model Derivation

- The matrix cover also impacts the number of input copies emitted by Mappers and the cost for shuffling them.

- We argue that those costs are sufficiently accounted for by the Reduce-phase analysis:
  - Map-phase duration differences for different matrix covers depend on the total number of input duplicates. Similarly, total shuffle time differences depend on the total number of input duplicates caused by a matrix cover.
  - All input duplicates are processed by Reducers, hence their impact is reflected there.

Reducer-Centric Cost Model

- Clearly, our Reducer-centric cost analysis abstracts away certain aspects, e.g., the impact of a slow network on shuffle time.

- Hence it is important to perform a thorough empirical evaluation on realistic data and hardware to prove the effectiveness of the approach.

Reducer-Centric Cost Model

- The advantage of a simple (yet sufficiently accurate) cost model is that it enables strong analytical results, e.g., optimality proofs. To get there, we make one more (slightly) simplifying assumption:

- More input a worker processes and more output it generates, the longer it will take. This will generally hold for theta-joins in practice, because it takes longer to match larger inputs and produce larger outputs.

Reducer-Centric Cost Model

- Putting things together, it now is clear that to minimize running time in this cost model, we need to minimize the amount of input and output assigned to each worker.

- We use max-input and max-output to refer to the largest input any worker receives and the largest output any worker produces, respectively.

Reducer-Centric Cost Model

- For simplicity and without loss of generality, we assume each worker receives at most 1 Reduce task. This way analyzing a Reduce task directly reveals input and output for the corresponding worker.

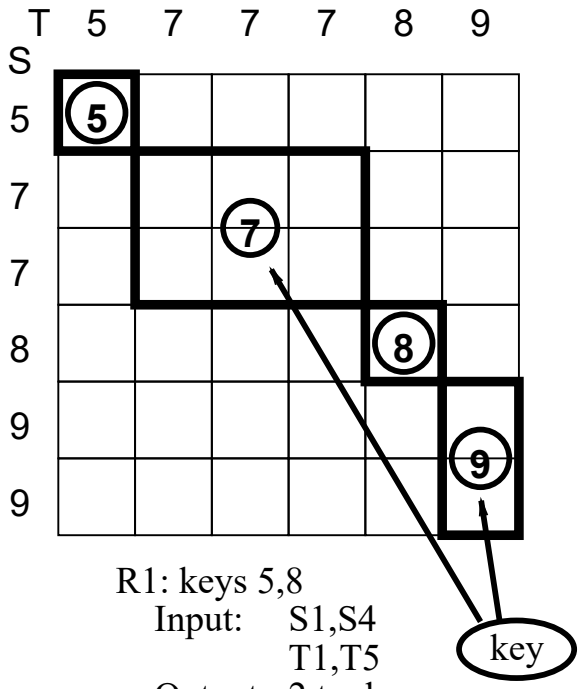
Let us look at example covers to better understand the tradeoffs.

Example

- The idea is that we want to perform an equi-join on tuples from S and T.

- Assume we have 3 workers for our Reduce phase.

Hash+shuffle Join:



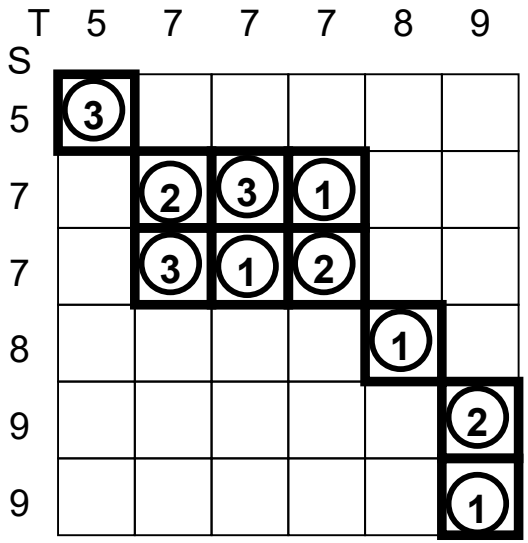
R1: keys 5,8
Input: S1,S4
T1,T5
Output: 2 tuples

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:



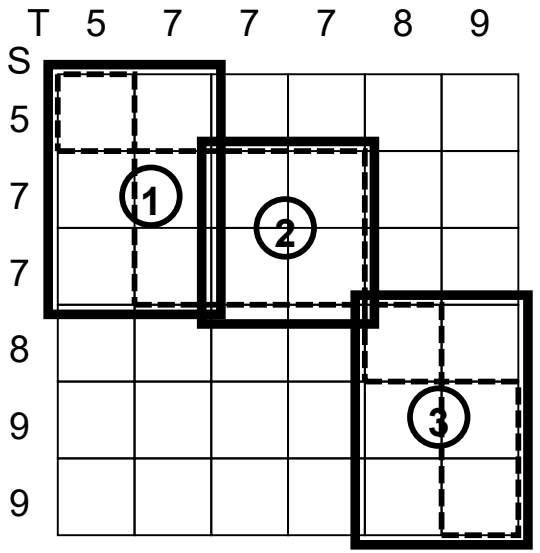
R1: key 1
Input: S2,S3,S4,S6
T3,T4,T5,T6
Output: 4 tuples

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

- Hash+shuffle:
  - Using the join attribute as key and resulting in 4 regions.
  - The only problem is load imbalance caused by data skew, in particular by the frequent occurrence of join value 7.

- Random assignment of “true”-valued cells:
  - Addressed skew by lowering max-reducer-output to 4, but increases max-reducer-input due to excessive duplication.

- Given regions/blocks:
  - Even though some “false”-valued cells are covered unnecessarily, the cover achieves max-reducer-input as low as hash+shuffle (since it avoids grouping by the join attribute), and at the same time max-reducer-output as low as the random assignment.

From Matrix Cover to Algorithm

- Before discussing how to find optimal matrix covers, we first establish how a cover can be converted to a distributed algorithm.

- To avoid the subtle output-duplication problem of the example cover in the center, we require key regions to be rectangles.

From Matrix Cover to Algorithm

- *If we cover the entire matrix* with non-overlapping rectangles, then a simple randomized 2-round algorithm can compute any theta-join:
  - Round 1 = each task receives the cover information. For an S-tuple s, it selects a random matrix row and sends a copy of s to all cover regions intersecting that row. (same for a random column for each T-tuple.)
  - The data is shuffled to group it by region. Round 2 = compute the join in each region separately. This local computation can leverage existing libraries of efficient (non-parallel) join implementations.
- We will refer to this algorithm as Basic-Theta.

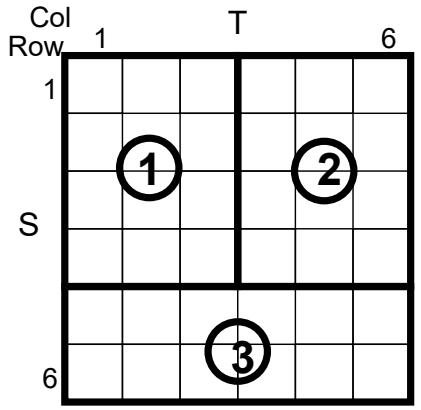
```
Class Mapper {
 // A Cover of the entire join matrix
 Cover

 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 x with their keys
   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 x with their keys
   for each regionID in Cover.getRegions( matrixCol )
    emit( regionID, (x, "T") )
  }
 }
}
```



Example cover

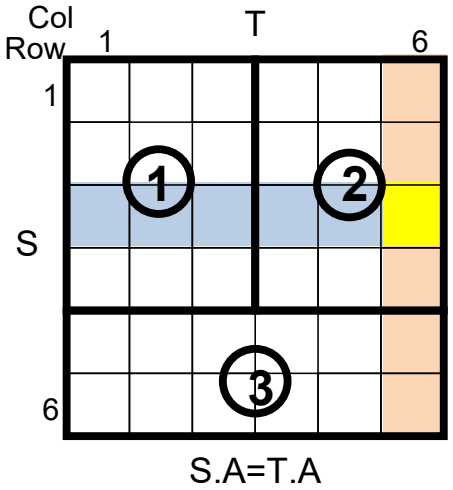
```
reduce( regionID, [(x1, flag1), (x2, flag2),…]) {
  initialize S_list and T_list

  // 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

- Since each cell is covered by exactly one key, it is easy to show that the algorithm correctly implements any theta-join.
- Consider output tuple (s_i, t_j). Inputs 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.



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:

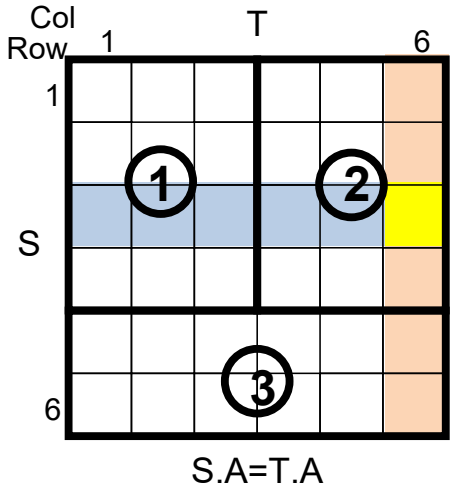
Reducer X: key 1
Input: S1, S3, S5 ,S6 T2, T3, T4
Output: (S3,T2),(S3,T3),(S3,T4)

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)

Algorithm Correctness

- That 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.



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)

Why Randomization?

- Reason 1: simpler algorithm
  - Mappers do not know the "correct" row of an input tuple. In the 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.

Why Randomization?

- Reason 2: performance
  - Randomization effectively addresses 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 produce an overly large share of the output. Randomizing rows and columns shuffles the “true”-valued cells around in the matrix, balancing load across regions.

Why Randomization?

- Reason 2: performance
  - 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.

Optimal Cover: Entire Matrix

- We established earlier that any cover of the entire join matrix using non-overlapping rectangles guarantees correctness for any theta-join implementation.

- However, there are many possible such matrix covers. Given r, the desired number of regions, our goal is to find the optimal cover.

Ideal Cover: What does a Rectangle Cost?

- Suppose one reducer receives one rectangular region.
  - Region "height" = number of S rows
  - Region "width" = number of T columns

- What is the input to reducer?

- What is the total pairs checked/output candidates?

Ideal Cover: What does a Rectangle Cost?

- Suppose one reducer receives one rectangular region.
  - Region "height" = number of S rows
  - Region "width" = number of T columns

- What is the input to reducer? **height + width**

- What is the total pairs checked/output candidates? **height x width**

Ideal Cover: What does a Rectangle Cost?

- So a rectangle has two relevant quantities:

- height + width: *how many input tuples are sent to the reducer*
-
- height x width: *how many candidate pairs the reducer may examine*

Ideal Cover: Squares

- Assume we have regions where both cover 16 cells.
- "Skinny": height = 1, width = 16
- Square region: height = 4, width = 4
- Which one has a smaller input cost?

Squares & Target Side Length

- C = target side length of one ideal region

- So ideally, each reducer gets a region like: C rows x C columns

- This means each reducer gets approximately: C S-tuples + C T-tuples as input
- C x C candidate cells as work

How to Choose C, Row and Column Groups

- Let A = number of row groups, B = number of column groups

- If the full matrix has: |S| rows, |T| columns and each region should have about C rows and C columns, then

$$A \approx \frac{|S|}{C}, B \approx \frac{|T|}{C}$$

Why Does the Ratio Appear?

- From

$$A \approx \frac{|S|}{C}, B \approx \frac{|T|}{C}$$

- We can divide the 2 equations and cancel the C's:

$$\frac{A}{B} \approx \frac{|S|/C}{|T|/C}$$

$$\frac{A}{B} \approx \frac{|S|}{|T|}$$

Why Does the Ratio Appear?

- **Intuition:** The grid should have the same shape as the matrix.

- If the matrix is wider than it is tall, then we need more column groups than row groups.

Example

- $|S| = 4000, |T| = 6000$
- Then: $|S| / |T| = 4000 / 6000 = 2/3$
- A natural choice is: $A = 2, B = 3$
- In words, that means: 2 row groups, 3 column groups, for a 2 x 3 = 6 regions

How do we Choose C?

- C is chosen so that about r square-like chunks cover the matrix.

- Now use the number of reducers. If we want about r regions, then: A x B approx r.

- Using: $$A \approx \frac{|S|}{C}, B \approx \frac{|T|}{C}$$

How do we Choose C?

- We get:

$$\frac{|S|}{C} \cdot \frac{|T|}{C} \approx r$$

$$\frac{|S||T|}{C^2} \approx r$$

- Therefore:

$$C^2 \approx \frac{|S||T|}{r}$$

$$C \approx \sqrt{\frac{|S||T|}{r}}$$

How do we Choose C? Example

- Given |S| = 4000, |T|=6000, r=6
  - Matrix size: 4000 rows × 6000 columns
  - We want 6 regions.

- The ideal area per reducer is: (|S|* |T|) / 6 = (4000 * 6000) / 6 = 4,000,000. So each ideal region should cover 4 million cells. A square-like region with 4 million cells has side length: C = 2000. Now compute:
  - A = |S|/C = 4000 / 2000 =2
  - B = |T|/C = 6000 / 2000 = 3

- So the cover is: A x B = 2 x 3 = 6.

What C x C Squares do not Tile?

- Example: r = 9, and T has 50% more tuples than S. This means that |S| / |T| = 2/3.

- The grid should preserve the same ratio: A / B approx 2 / 3. A natural grid is: A = 2 and B = 3.

- So: A × B = 6 regions. Even though we had r=9 reducers available, only 6 regions are used.
  - The regions are larger than the ideal C x C regions, but they cover the matrix cleanly and remain reasonably square-like.

1-Bucket-Random

- Basic-Theta needs a rectangular cover. 1-Bucket chooses that cover using only |S|, |T|, and r. Goal: Use at most r reducers and make regions square-like.

- Let: A = row groups, B = column groups, ρ = |S| / |T| assuming |S| <= |T|
  - If |S| < |T|, the matrix is wider than it is tall.

- Choose A and B so that: A x B = r (number of regions), A / B approx ρ (same shape as the matrix)

1-Bucket-Random

- So: A approx sqrt(ρ x r), B approx sqrt(r / ρ).

- If S is tiny (T is more than r times bigger than S): ρ < 1/r, then this implies A = 1, B = r.
  - What does this remind us of?
  -
  -
- Otherwise: A = floor(sqrt(ρ x r)), B = (sqrt(r / ρ)). Then run Basic-Theta on the A × B grid.

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 compute A and B 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.

1-Bucket Analysis: Cartesian Product

- 1-Bucket relies on the matrix cover used in the analysis. 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.

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...

Matrix Multiplication

- Linear algebra is an important mathematical tool for data analysis. Equations in linear algebra are naturally expressed as manipulations of matrices and vectors.

- Problems such as finding paths in a graph and computing PageRank can be expressed as matrix multiplication problems.

Matrix Multiplication

- 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.

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

Parallel Matrix Multiplication: Row-by-Column

- Row-by-Column Partitioning: A by rows and B by columns. Then every A row-block must be combined with every B column-block. This creates a work matrix:

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

Here each A_i and B_j is a matrix, containing some of A's rows and B's columns, respectively. Each product $A_i B_j$ can be computed independently.

Parallel Matrix Multiplication: Row-by-Column

- Each cell is one independent output block. Connection to 1-Bucket:
  - A row-blocks behave like S-tuples.
  - B column-blocks behave like T-tuples.
  - Each reducer owns a rectangle of this work matrix.

- In theta joins, the matrix cells were candidate pairs (s,t). Here, the matrix cells are output blocks Ai*Bj.
  - The same matrix-cover idea applies, but now the reducer computes a matrix product instead of testing a predicate.

Parallel Matrix Multiplication: Row-by-Column

- 2 row blocks of A, 2 column blocks of B

	B1	B2
A1	R1	R2
A2	R3	R4

- Routing: A1 -> {R1, R2} A2 -> {R3, R4} B1 -> {R1, R3} B2 -> {R2, R4}

- Reducer work: R1 computes A1 x B1, R2 computes A1 x B2, R3 computes A2 x B1, R4 computes A2 x B2.

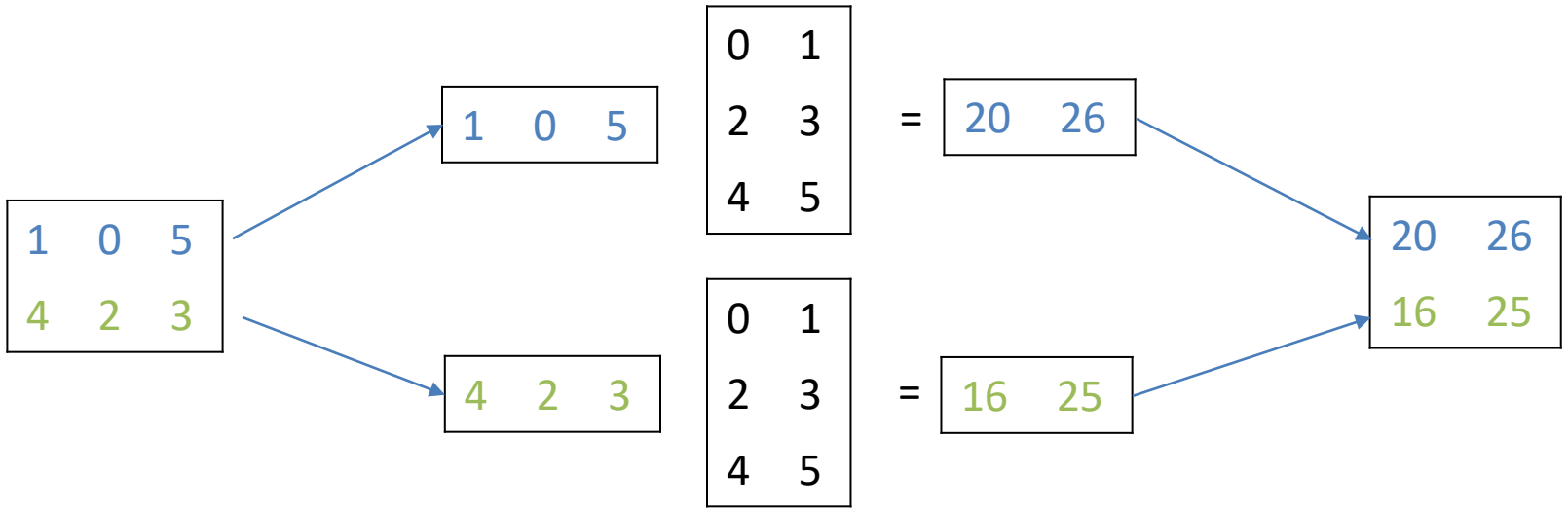
Example for Row-by-Column Partitioning

Consider ***AB*** example

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

Leave ***B*** as is, but partition ***A*** into A_1 = [1 0 5] and A_2 = [4 2 3]

Terms A_1B and A_2B produce result matrix rows [20 26] and [16 25]



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!

- Standard matrix multiplication computes **one output cell at a time**. Column-by-row partitioning computes **one contribution layer at a time**.

Parallel Matrix Multiplication: Column-by-Row

- Suppose A is m x p, and B is p x n. The result is then C = AB (which is m x n).

- Each output cell is:

$$C[i,j] = A[i,1]B[1,j] + A[i,2]B[2,j] + \ldots + A[i,p]B[p,j]$$

- So every output cell is a **sum over the shared inner index**.

Parallel Matrix Multiplication: Column-by-Row

- The usual way says: To compute C[i,j], take row i of A and column j of B, multiply matching positions, then sum.

- The alternative way says: For each k, take column k of A and row k of B. Their product creates the k-th contribution to **every cell** of the output matrix.

- This is called an **outer product** view.

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***:

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

Matrix sum ⊕

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

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.

```
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 )
}
```

Data Transfer Tradeoff: Row-By-Col (1-Bucket)

- Partition A into G row groups. Partition B into H column groups.
- Each A block is needed by all H column groups. Each B block is needed by all G row groups.
- Map-to-Reduce transfer: $H|A| + G|B|$
- Benefit: reducers directly compute final output blocks. Cost: input blocks are duplicated during shuffle.

Data Transfer Tradeoff: Col-By-Row

- Partition A by columns. Partition B by rows.

- First MapReduce job: each A entry is sent once and each B entry is sent once. First shuffle = |A| + |B|

- But reducers emit partial products. A second MapReduce job groups those partial products by output cell.

Data Transfer Tradeoff: Example

- Suppose: A is 1000 x 100 and B is 100 x 1000
- So: |A| = 1000 * 100 = 100,000 entries, and |B| = 100 * 1000 = 100,000 entries
- The output matrix C = AB is: 1000 x 1000 = 1,000,000 output cells

Data Transfer Tradeoff: Example (1-Bucket)

- Suppose we partition A into G = 10 row groups, and B into H = 10 column groups.

- Then we have:
  - G * H = 10 * 10 = 100 reducers / output blocks

- Each A row block is sent to all 10 column groups:
  - A transfer = H|A| = 10 * 100,000 = 1,000,000 entries

Data Transfer Tradeoff: Example (1-Bucket)

- Each B column block is sent to all 10 row groups:
  - B transfer = G|B| = 10 * 100,000 = 1,000,000 entries

- Total Map-to-Reduce transfer:
  - 1,000,000 + 1,000,000 = 2,000,000 entries

- After that, each reducer computes its assigned output block directly. No second shuffle is needed just to sum partial products.

Data Transfer Tradeoff: Example (Col-By-Row)

- Each A entry is sent once, and each B entry is sent once. T he first shuffle: |A| + |B| = 100,000 + 100,000 = 200,000 entries

- That looks much cheaper initially. But now consider the partial products. For each shared index k, the reducer takes: column k of A: 1000 values and row k of B: 1000 values.
  - It produces: 1000 * 1000 = 1,000,000 partial products

Data Transfer Tradeoff: Example (Col-By-Row)

- There are 100 possible k-values, so total partial products: 100 * 1,000,000 = 100,000,000 partial products.

- Those partial products must be shuffled in the second job so that all contributions for the same output cell (i,j) meet and get summed.

- So total map-to-reduce transfer is approximately: first shuffle (200,000 entries) + second shuffle (100,000,000 partial products). The total is 100,200,000 transferred records

Multiplying a Matrix by its Transpose

- Assume A is an objects x features matrix.
  - Example: rows = documents
  - columns = words
  - values = counts or weights

- Then A x A^T is an objects x objects matrix.
  - Entry (i, j): $(AA^T)[i,j]$ = row i of A * row j of A
  - Interpretation: AA^T compares every object with every other object. For documents: $(AA^T)[i,j]$ is the similarity between document i and document j.

Multiplying a Matrix by its Transpose

- Distributed idea: process one feature/column at a time. For each column k, look at all rows that have a value in column k

- For every pair of such rows (i, j):
  - emit partial contribution:
    - key: (i, j)
    - value: A[i,k] * A[j,k]

- Reduce: sum all values for each (i, j). Result: $(AA^T)[i,j]$

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]AT[k,*] can already be computed in the Mappers, letting the Reducers perform the final aggregation step. This eliminates the additional post-processing phase.

```
// 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.

Matrix Multiplication in Machine Learning

- Many other machine learning techniques can be implemented using matrix products, including:
  - Locally Weighted Linear Regression
  - K-means clustering
  - Logistic Regression
  - Neural Network (for backpropagation)
  - Principal Component Analysis
  - Independent Component Analysis
  - Support Vector Machine (SVM) with linear kernel

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.

References

- A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In Proc. ACM SIGMOD Int. Conf. on Management 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*, 37(3): 411-439, Springer, 2019
  - http://www.ccs.neu.edu/home/mirek/papers/2019-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=10102065753484484478&hl=en&as_sdt=0,22