CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 14, 5/14/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

14 Outline

- Map reduce and indexing
- Sparse matrix multiplication using SQL
- Joins using map reduce

14.1 Map Reduce

Recall what map reduce does: it's a tool for putting pairs that have the same key together. The map reduce environment gave the following promise:

- Map phase: emits pairs of the form $\langle \text{key}, \text{value} \rangle$ (shuffle).
- Shuffle phase: places pairs with the same key on a single machine.
- Reduce phase: performs computations on pairs with the same key.

On the face of it, this may look trivial, however, this construct is powerful. If your write your code according to this contract you get fault tolerance, you get distribution, and utilization of all cores in the cluster.

Example Suppose we've indexed the web by crawling it with a thousand machines. So now each machine has stored on it a part of the web; each machine is full of HTML and pointers to other HTML pages. From this, we'd like to build a search engine kind of like Google. So, how can we apply map reduce to a search engine? Given a key word, we need to be able to *very* quickly figure out which web-pages have that key in them; this is the most basic search engine functionality. If we have an inverted-index, we can do this efficiently.

An inverted-index is a list of words, where each word is associated with the pages it occurs in. So, we'd like to get to a point where we have a list

$$\text{word}_i \to \{\text{page}_1, \text{page}_2, \ldots\}$$

How can we create such an index using map reduce? All we have is a jumble of web-pages, and we want a map between the word itself and the pages on which it occurs. We only have one hammer at our disposal, map-reduce, which puts pairs with the same key together. So, naturally, the key should be the word itself, and the value should be the url.

```
// Map phase
for $w \in H$ do
    emit $(w, h_{uid})$
end
// Reduce phase
Reduce(w, $\langle \mathrm{url}_1, \mathrm{url}_2, \ldots, \rangle$)
```

Algorithm 1: Mapper

So, each mapper will examine the web-page that it has been given. For each word in the web-page, emit the word as the key and the url as the value.

The above code is low-level, and can be tedious to write ourselves. Instead, we move onto SQL, and specifically multiplying sparse matrices.

14.2 Sparse Matrix-Vector Multiply using SQL

First of all, let's figure out how our matrix is represented. Since our matrix M is sparse, we need the following representation

$$(i, j, \text{value})$$

i.e. for each non-zero entry we store the row and column indices alongside the corresponding value. A sparse vector x is represented as (i, value).

Now, imagine that our matrix is an SQL table, where the columns in the table is i, j, and the value; similarly for our matrix, represented as a table with a column for the i index and a column for the value.

How can we compute Mx using SQL operations?[1] We realize that if $Mx = b$, then each entry of b an inner product between corresponding row of M and vector x. But how can we prepare these dot-products? We first perform a join on M and x where we join M on its column index j and x on its element index i. Our resulting table looks like

$M.j(x_i)$	$M.i$	M.value	x.value

Notice that where there are duplicates in our join, a cartesian product will appear. I.e. for every single entry in matrix M, we will get the corresponding value from vector x.

M =

i	j	value
1	2	17.5
2	2	16.3

x =

i	value
1	2.71
2	3.14

[1] First attempt: We can first group by i on matrix table M; at this point, each entry in the resulting table will correspond to a row in M, where the value is a function of the values in the columns of M for said row. When we do a `group-by`, we need to specify what kind of aggregation function we perform to combine the values for a particular group. What kind of aggregation should we use? We abandon this approach.

The resulting table after our `inner-join`

$M.j(x_i)$	$M.i$	M.value	x.value
2	1	37.5	3.14
2	2	18.3	3.14

The second SQL statement simply adds a new column onto this table. Thew column will look like

$M.j(x_i)$	$M.i$	M.value	x.value	mult
2	1	37.5	3.14	37.5 × 3.14
2	2	18.3	3.14	...

We've done the multiplication in each dot-product. Now we need to perform the summation, i.e. perform a group by i and sum over our column `mult` which we added above. We'd like to take all of the multiplications that are associated with a particular dot-product and sum them up, so our aggregation function is a summation on `mult`. Notice that we only need as output the index i and corresponding value of Mx, so we can remove other columns.

Now that we've done this in SQL, how can we do this in map-reduce?

14.3 SQL Group-by using map reduce

We first implement a `group-by`, which groups all rows that have the same key and performs an aggregation function on remaining columns. A map reduce is just that. To implement a `group-by` using map reduce, all we have to do is

```
emit the group-by column as key
perform desired aggregation function on non-group-by columns in the reducer
```

Algorithm 2: group by using map reduce

Let's walk through an example. Suppose we have the following table.

a	b
1	1
1	2
3	3
4	5
5	11

We wish to `group-by` a and sum elements in column b. How can we do this in map reduce? Each row assumed to be an input to a mapper.

```
Group-by a
   sum(b)
```

So, a map reduce is essentially a group-by; they're effectively the same.

14.4 SQL (inner) join using map reduce

This is much more difficult than a `group-by`. There are several cases:

- both tables are so large that neither will fit on a single machine, and
- one table is small enough to fit locally.[2]

The implementations for these two cases are very distinct. In our join, we need to respect the duplicity of the keys. I.e. we must perform the cross-product with all entries appearing in the other table with the same key.

The case when one table fits in memory Suppose we want to merge tables T_1 and T_2, where table T_1 is small enough to fit in memory. For the sake of discussion, let's suppose that each table has two columns.

$$T_1 = \begin{array}{cc}\hline \text{a} & \text{b} \\ \hline\end{array}, \qquad T_2 = \begin{array}{cc}\hline \text{i} & \text{j} \\ \hline\end{array}$$

Do we even need a cluster to perform this? Yes, we definitely need a distributed environment, since the resulting join could be larger than table B. But do we need to perform our sorting operation on our tables? No, we will see that we actually don't need this.

We first broadcast T_1 to all machines via a bit-torrent broadcast. Now, each machine has a full-on copy of table T_1. For fast look-up, we place T_1 into a hash-table.

```
Place $T_1$ in a hash-table for fast look-up
Place hashed $T_1$ on each machine via a broadcast
Perform an inner join of whatever data in $T_2$ is stored locally on each machine with its
  local copy of (hashed) $T_1$
```

Algorithm 3: (Hash or Broadcast) Inner Join

Each machine does a join of what it owns from T_2 with *all-of* T_1. We don't even need to communicate this result to other machines, we can just leave the merged data local. The inner join that is performed locally on each machine can be done just as we did on the midterm. We assume that the resulting table can be fit on each machine; in the event that we have a cross-product which blows up our table, we would just start assuming that T_1 so large it can't fit on a single machine. We cover this case next, i.e. the case where neither table can fit locally.

14.5 Computations on Matrices with Spark

14.5.1 Distributed Matrices

In Spark, matrices are typically stored broken up for storage in three different ways:

[2] A third case is actually that both tables can fit on a single machine. This is covered by our PRAM model, and we saw this exact question on the midterm.

- By entries (CoordinateMatrix): stored as a list of $(i, j, value)$ tuples
- By rows (RowMatrix): each row is stored separately (e.g. Pagerank)
- By blocks (BlockMatrix): by storing submatrices of a matrix as dense matrices, block matrices can take advantage of low-level linear algebra library for operations like multiplications.

14.5.2 RowMatrix × LocalMatrix

When multiplying a RowMatrix with a small local matrix, we broadcast the entire small matrix to each machine that contains different parts of the RowMatrix and perform multiplications on each machine. Currently, Spark uses BLAS level 1 optimization, which optimizes for vector-vector multiplications during the multiplication.

$$\text{rows are distributed}\left\{\begin{bmatrix} \text{—} & r_1^T & \text{—} \\ \text{—} & r_2^T & \text{—} \\ & \vdots & \\ \text{—} & r_n^T & \text{—} \end{bmatrix}\right. \begin{bmatrix} | & | & & | \\ l_1 & l_2 & \dots & l_m \\ | & | & & | \end{bmatrix}$$

14.5.3 CoordinateMatrix × CoordinateMatrix

CoordinateMatrix is used to represent sparse matrices, and is stored as a list of (row, column, value) entries. To perform matrix multiplication of two Coordinate Matrix elements $C = AB$, one can summarize the procedure as follows:

```
input: $A : \{(i, j, A_{ij}) | A_{ij} \neq 0\}, B : \{(i, j, B_{ij}) | B_{ij} \neq 0\}$
$J \leftarrow$ Join $A, B$ on $a.j$ and $b.i$ for $a \in A, b \in B$
$M \leftarrow$ For each $j \in J$, map it to (key, value) where key=$(a.i, b.j)$ and value=$a.val \times b.val$
$C \leftarrow$ Reduce $M$ with "+"
```

Algorithm 4: CoordinateMatrix Multiplication

Unfortunately, Spark does not have CoordinateMatrix multiplication implemented in the current library. One possible implementation with Scala, when assuming the matrices are stored as `RDD[MatrixEntry(i, j, value)]` is shown below

```
    import org.apache.spark.mllib.linalg.distributed._

val n = 10          // one dimension of the matrix
val range = sc.parallelize({1 to n * n})
// Generate two random sparse matrices of size nxn
val A = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))
val B = range.sample(false, 0.2).map(i => MatrixEntry(i / n, i % n, i))

// Perform multiplication
```

```
val C = A.map(e => (e.j, e)).join(B.map(e => (e.i, e)))
         .map(p => ((p._2._1.i, p._2._2.j), p._2._1.value * p._2._2.value))
         .reduceByKey(_ + _).map(p => MatrixEntry(p._1._1, p._1._2, p._2))
```

Effectively, for each $1 \leq i \leq n$, we first join all entries of matrix A on the i-th column with the entries of matrix B on the i-th row, which would create a Cartesian product of two sets of entries for each i. The resulting set contains all possible pairs of entries that would've been multiplied together during a normal matrix multiplication, and each pair of entries is keyed by their shared dimension during the dot product operation. We then remap each element in this set by its position in the result matrix and change its value to the product of the two entries and then reduce each result position with the addition operator. This effectively simulates the dot product operation. Finally, we remap the result to the desired format.

14.5.4 BlockMatrix × BlockMatrix

In some cases, we'd like to multiply two dense matrices for which the rows and columns may themselves be too large to fit in memory on a single machine. By partitioning our matrices into blocks that do fit on a single machine - encoding each one as a BlockMatrix - and performing multiplication on their partitions, we can manage to perform matrix computation on the larger matrices. Using BlockMatrix, we also have the ability to push down the smaller block matrix multiplications to the CPU/GPU directly using Basic Linear Algebra Subprograms (BLAS) routines - as mentioned above, Spark currently uses BLAS level 1 for matrix multiplication. To perform block matrix multiplication, we partition both matrices appropriately so their blocks are equally sized within each matrix, aligned in size across matrices, and so that a single block from each matrix fits together on a single machine. Following partitioning, block multiplication proceeds similarly to coordinate multiplication. Each matrix is flatmapped to produce a list of blocks for multiplication - each block in the first matrix A is effectively copied as many times as the number of columns in the second matrix B, and each block in B is effectively copied as many times as the number of rows in A. Following the flatmap, a cogroup is used to send pairs of complementary blocks that will need to be multiplied to an individual machine, where the multiplication is pushed down to the CPU/GPU level using BLAS. Finally, results of the individual block multiplications corresponding to each block entry in the resulting matrix are sent to the same machine with ReduceByKey and summed up. A simplified version of the Spark code for block matrix multiplication is presented below:

```
def multiply(other: BlockMatrix): BlockMatrix = {

      // Get partitions
      val resultPartitioner = GridPartitioner(numRowBlocks, other.numColBlocks,
        math.max(blocks.partitions.length, other.blocks.partitions.length))

      // Each block of A must be multiplied with the corresponding blocks
      // in each column of B.
      val flatA = blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(other.numColBlocks)
                (j => ((blockRowIndex, j, blockColIndex), block))
      }

      // Each block of B must be multiplied with the corresponding blocks
      // in each row of A.
      val flatB = other.blocks.flatMap {
        case ((blockRowIndex, blockColIndex), block) =>
            Iterator.tabulate(numRowBlocks)
                (i => ((i, blockColIndex, blockRowIndex), block))
      }

      // Cogroup and multiply block pairs
      val newBlocks: RDD[MatrixBlock] = flatA.cogroup(flatB, resultPartitioner)
        .flatMap { case ((blockRowIndex, blockColIndex, _), (a, b)) =>
          if (a.nonEmpty && b.nonEmpty) {
            val C = b.head match {
              case dense: DenseMatrix => a.head.multiply(dense) // Uses BLAS 1
              case sparse: SparseMatrix => a.head.multiply(sparse.toDense)
            }
            Iterator(((blockRowIndex, blockColIndex), C.toBreeze))
          } else {
            Iterator()
          }
      }
      // Sum up matrices for each block entry of C
      .reduceByKey(resultPartitioner, (a, b) => a + b)
        .mapValues(Matrices.fromBreeze)
}
```