CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 10, 4/29/2015. Scribed by Jan Dlabal, Fang-Chieh Chou, Yu-Wei Lin, Yi-Hong Kuo.

10 Covariance Matrices and All-pairs Similarity

10.1 Some Quick Review

- **Shuffle size** is the size of all pairs emitted by all mappers.
- **All to all communication** will be kicked off by:
  - groupByKey,
  - join,
  - reduceByKey, or
  - repartition.
- **All to one** communication will be performed by:
  - reduce, if done well, or
  - broadcast.

10.2 Introduction

In statistics and data mining, sometimes we would like to calculate A^TA for large $m \times n$ matrix A. For example, the matrix A is the movie rating matrix where each rows is a user's rating for different movies. To give recommendation, we need to calculate "similarity" between all pairs of columns in the matrix. Calculating all-pairs similatiry is essentially the same computation for A^TA, which entry (i, j) holds dot product $(c_i^Tc_j)$ of column c_i and c_j of A. Therefore, the goal in this lecture is to develop algorithm to compute A^TA in a distributed way.

10.3 Assumptions for the matrix A

Assume A is an $m \times n$ matrix with the following properties:

- The matrix is tall and skinny ($m \gg n$). For example, $m = 10^{12}$ with $n = 10^4$ or 10^6.
- Rows are sparse, with at most L non-zero entries in each row.
- The matrix is stored row by row and can be accessed by row index. For example, each row is an entry of the RDD, which can fit into memory on a single machine.
- A^TA is considerably much smaller than A ($n \times n$, where n is small), and is a dense matrix.

10.4 Naive Algorithm

Since $A^TA = \sum_{i=0}^{m} r_i r_i^T$, where r_i is i-th row of matrix A, the naive way to calculate A^TA is to implement the above calculation (slide p.8).

Algorithm 1 Naive Implementation

```
procedure NaiveMap(r_i)                ▷ emit matrix r_i r_i^T, the outer product of r_i by itself.
    for all all pairs (a_ij, a_ik) in r_i do
        Emit ((j, k), a_ij a_ik)
    end for
end procedure

procedure NaiveReduce(((i, j),⟨v_1, ..., v_R⟩))        ▷ sum up all the matrices
    output c_i^T c_j = Σ_{k=1}^{R} v_k
end procedure
```

Here the mapper emits at most L^2 numbers per row, so the shuffle size is bounded by $O(mL^2)$.

For the reducer, note that two columns might be identical and dense. For example, in the Netflix case, a large number of users might have seen both Godfather and Godfather 2. This is why in the worst case we'll have to do $O(m)$ work in each reduce step, where it needs to sum over at most m numbers. When m is large, the naive algorithm may become intractable.

10.5 DIMSUM: Dimension Independent Matrix Square Using MapReduce

DIMSUM provides a tractable way to compute the all-pairs similarity efficiently in distributive setting. The algorithm is shown below:

Algorithm 2 DIMSUM Implementation

```
procedure DimsumMap(r_i)
    for all all pairs (a_ij, a_ik) in r_i do
        With probability min(1, γ 1/(||c_i|| ||c_j||))
      Emit ((j, k), a_ij a_ik)
    end for
end procedure

procedure DimsumReduce(((i, j),⟨v_1, ..., v_R⟩))
    if 1/(||c_i|| ||c_j||) > 1 then
        output b_ij = 1/(||c_i|| ||c_j||) Σ_{k=1}^{R} v_k
    else
        output b_ij = (1/γ) Σ_{k=1}^{R} v_k
    end if
end procedure
```

DIMSUM Mapper is identical to the naive algorithm, except we emit with a probability instead of total sum (slide p.10). This probabilistic emission scheme down-samples dense columns to reduce the computational costs, with only minor impact on the accuracy of the similarity computed. DIMSUM Reducer will sum just as before, except it will have to scale the result given the probabilities used by the mapper. The emitted probability is controlled by a tunable parameter γ. Different values of γ gives different computational costs and errors, as we will show in the analysis below.

Here we are calculating cosine similarities instead of dot products. We also need to have the norm of every column pre-computed before applying DIMSUM, but computing column norms is much cheaper than computing dot products, so that's not a big deal to use in the probabilities, and the computation cost of norm won't affect our analysis below.

10.5.1 Analysis for DIMSUM

Intuition : we down-sample column-pairs that are both dense (i.e. having high column norms). In the naive algorithm, these dense column pairs will contribute a large amount of values to be summed in the reducer. With smart down-sampling, we can avoid the maximum $O(m)$ work per reducer.

Shuffle size The proof for the bound of shuffle size is give in slide p.12-13, and we won't repeat it here. A few clarification is given below.

First, the proof assumes the elements of the input matrix has only 0 and 1. In this case, the slide employs the notations $\#c_i$ and $\#(c_i, c_j)$. For 0/1 matrices, the column norms are $||c|| = \sqrt{\#(c)}$, where $\#(c)$ is the number of non-zero entries in the column c. Similarly, we use $\#(c_i, c_j)$ to represent the number of co-occurred non-zero entries between columns c_i and c_j, which simply equals to $c_i^T c_j$ in 0/1 matrices.

Because every row has at most L non-zeros (we assume the matrices have sparse rows), we have $\#(c_i, c_j) \leq L\#(c_i)$; this allows us do the last step on slide p.13. Therefore DIMSUM gives a much smaller bound that does not depend on m and is only linear in L, i.e. $O(\gamma Ln)$.

For a non-0/1 matrix, the running-time bound gets worse ($O(\gamma Ln/H^2)$, where H is the smallest nonzero entry in magnitude, slide p.15). For the 0/1 case, $H = 1$, but in the worst case scenario H can be quite small. For example, imagine the case where the matrix contain mostly small numbers ϵ's, and one huge number. In this case H can be arbitrarily small, leading to a bad running time.

In the case of 0/1 matrices, since DIMSUMReducer produces a number between 0 and 1, clearly the sum process only receives at most γ elements with the same key, (slide p.16).