JTSQR: An Implementation of TSQR in Hadoop MapReduce

Directory

JTSQR: An Implementation of TSQR in Hadoop MapReduce

1. Introduction

JTSQR is package that implements Tall-and-Skinny QR Factorization in Hadoop MapReduce using JAVA. The purpose of this report is to make users understand how to operate JTSQR more clearly. It presents some implementation details of JTSQR, such as package organization, QR Factorization of JTSQR, etc. Some performance results of JTSQR in are also included in the experiments and results section.

2. Implementation of JTSQR

This section presents the implementation of JTSQR package, including package organization and the TSQR algorithm.

Package organization of JTSQR

The codes of JTSQR include four MapReduce Jobs: subMatrixCountJob, subMatrixGenerateJob, QRFirstJob, BuildQJob. We refer them as Job1-1, Job1-2, Job2, and Job3 respectively, whose flow diagram is presented in Figure 1.

- Job1:
  Turn text file into Hadoop Sequencefile.
  - Job 1-1:
    Compute the number of submatrices depending on the argument "subRowSize" (row size of sub matrix), and give each sub matrix a unique sequential number. These data are recorded in "Matrix Count List".
  - Job 1-2:
    Give a number to each sub matrix in "Matrix Count List", turn text file into Hadoop Sequencefile, and output these sub matrices.
- Job2:
  Perform QR factorization of submatrices, and output "first level Q matrices" and final R. The "first level Q matrices" are a collection of Q matrices computed in

the TSQR algorithm, whose detail will be explained in section 2.

- Job3:
  Multiply these first level Q matrices and compute the final Q.

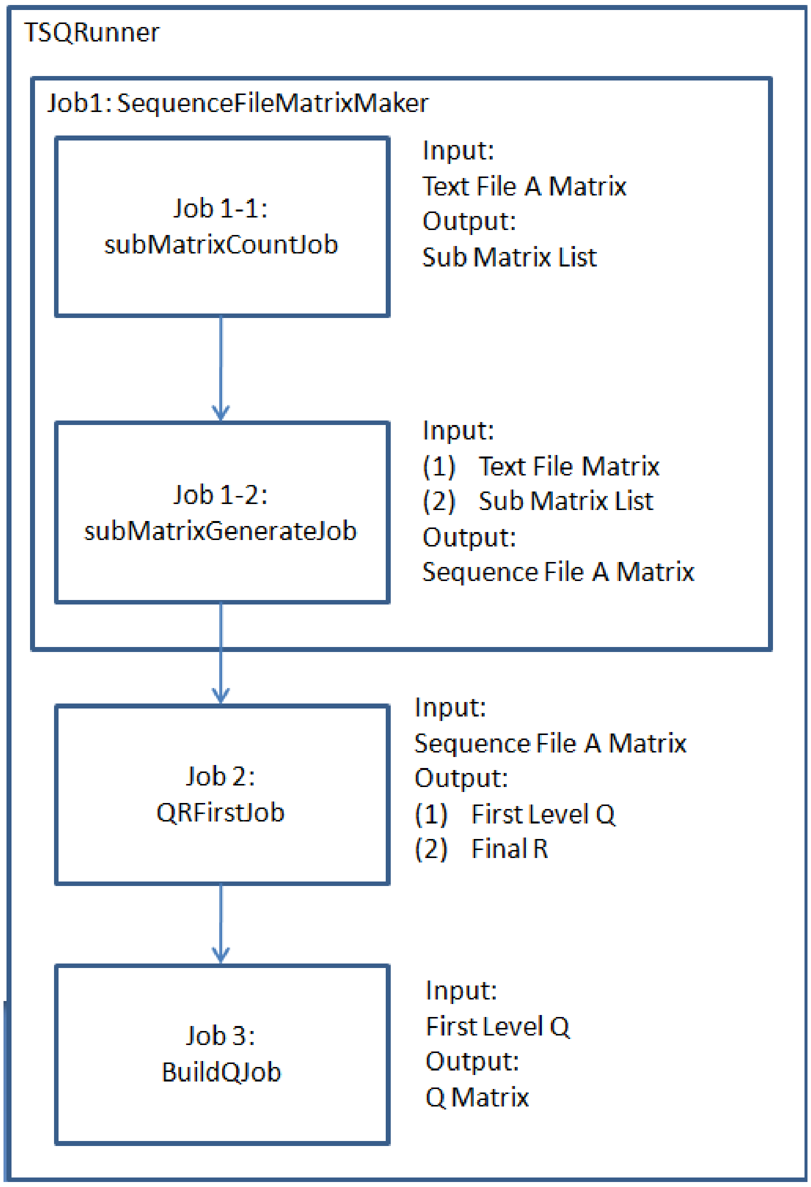
TSQRunner
Job1: SequenceFileMatrixMaker
Job 1-1: subMatrixCountJob
Input: Text File A Matrix
Output: Sub Matrix List
Job 1-2: subMatrixGenerateJob
Input: (1) Text File Matrix (2) Sub Matrix List
Output: Sequence File A Matrix
Job 2: QRFirstJob
Input: Sequence File A Matrix
Output: (1) First Level Q (2) Final R
Job 3: BuildQJob
Input: First Level Q
Output: Q Matrix

Figure 1 JTSQR Flow Diagram: Compute Q and R matrix

2.2. QR Factorization of JTSQR

The second job, FirstQRJob, implements the TSQR algorithm, whose steps are explained as follows. In the beginning, each Map Task reads a part of sub matrices A and then merges them into to one matrix, called "merged matrix". Next, a regular QR factorization is performed for the merged matrix, and the Q-factor and R-factor are output. In the Reduce phase, each Reduce Task merges some previously output R-factor and performs QR factorization on it. The MapReduce iteration is repeated until the number of Reduce Task becomes one. Finally, the final R matrix and all intermediate Q matrices are output. Figure 2 shows this work flow.

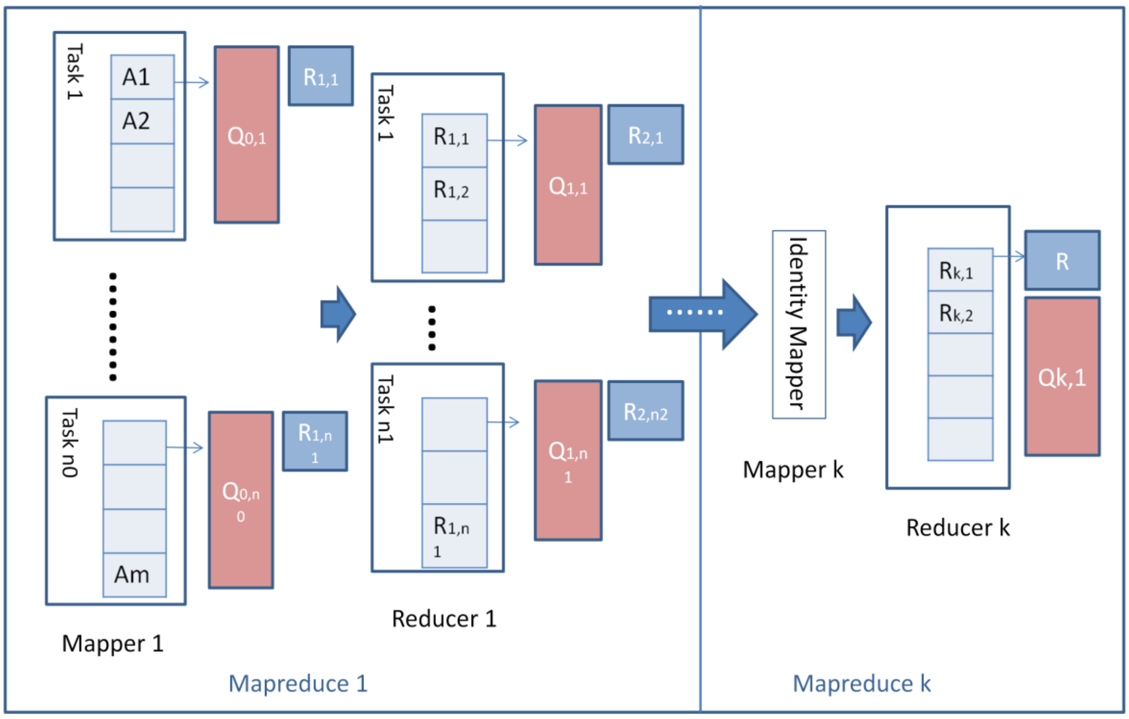

Figure 2 Job2 – Workflow of FirstQRJob

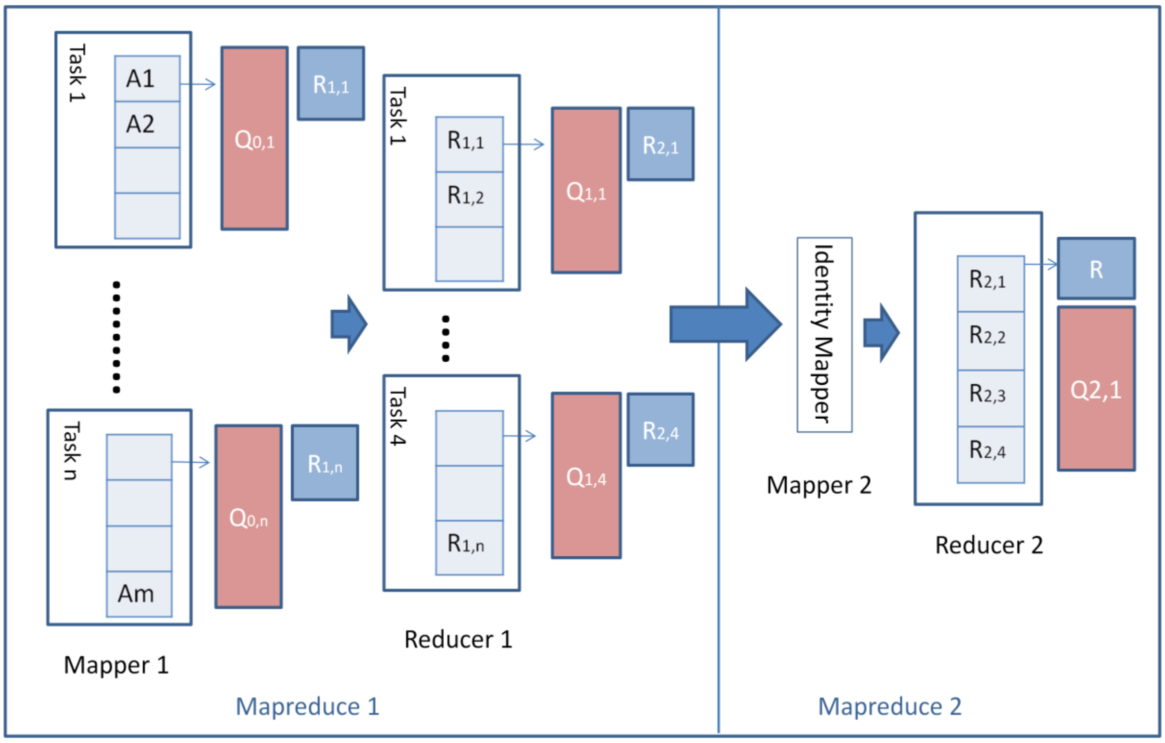

Figure 3 FirstQRJob schematic diagram of "reduceSchedule 4,1" example.

The argument "reduceSchedule" decides the number of MapReduce iterations and the number of Reducers in each iteration. For example, “4,1” means that the FirstQRJob has two MapReduce iterations, because the argument has two numbers, separated by a comma. The first iteration has four Reducers and second iteration has one Reducer. Figure 3 shows the example of QRFirstJob that uses argument "reduceSchedule 4,1". The default value of "reduceSchedule" is 1.

After QRFirstJob, the next Job (BuildQJob) computes the final Q matrix by multiply these first level Q matrices.

3. Experiments and Results

This section presents two experimental results of JTSQR. Section 3.1 lists the hardware and software specification of the experimental platform. Section 3.2 compares the performance of JTSQR and DTSQR (Direct TSQR [1]), another MapReduce implementation TSQR algorithm using Python. Section 3.3 shows the performance influence of argument “reduceSchedule”.

3.1 Specs

All the experiments are performed on a Hadoop cluster that has 15 machines as data nodes. Each node is equipped with a dual socket CPU, Intel Xeon X5670, which has 6 cores running in 2.9 GHz. The memory in each node is 96GB. The cluster uses InfiniBand for cluster network.

Our operating system is CeontOS 6.4; Java version is 1.6.0_45; Hadoop version is 1.2.1; Python version is 2.6.6; Matrix-toolkits-java version is 0.9.9 (mtj-0.9.9) [3]. We used LAPACK and BLAS library in mtj package for matrix computations.

3.2 Performance Comparison between JTSQR and DTSQR

In this experiment, we compare the performance of JTSQR and DTSQR using different matrix sizes. The results are show in Figure 4 and Figure 5. In Figure 4, we fixed the column size to 100 and varied the row size from one hundred thousand (100,000) to sixteen hundred thousand (1,600,000). In Figure 5, the column size is fixed to 10, and the row sizes are varied from one million (1,000,000) to sixteen million (16,000,000). As can be seen, JSTQR is much faster than DTSQR in both cases. It should note that we do not include the time of transforming text file into Hadoop sequencefile, because this is not related to the TSQR algorithm.

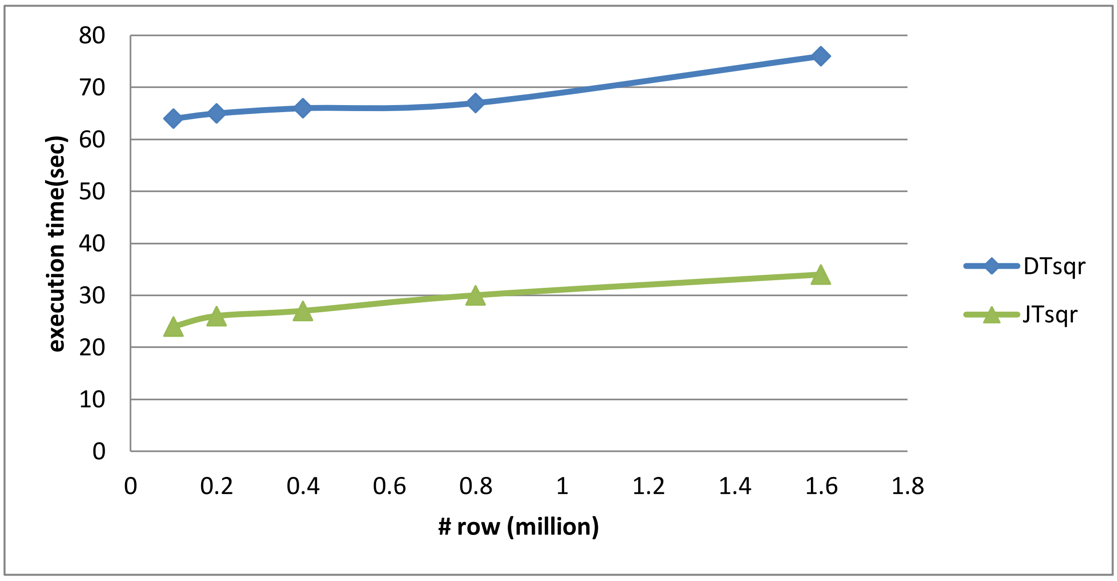

Figure 4 Performance comparison between JTSQR and DTSQR, the column size of matrices is 100, "reduceSchedule" of JTSQR is "1".

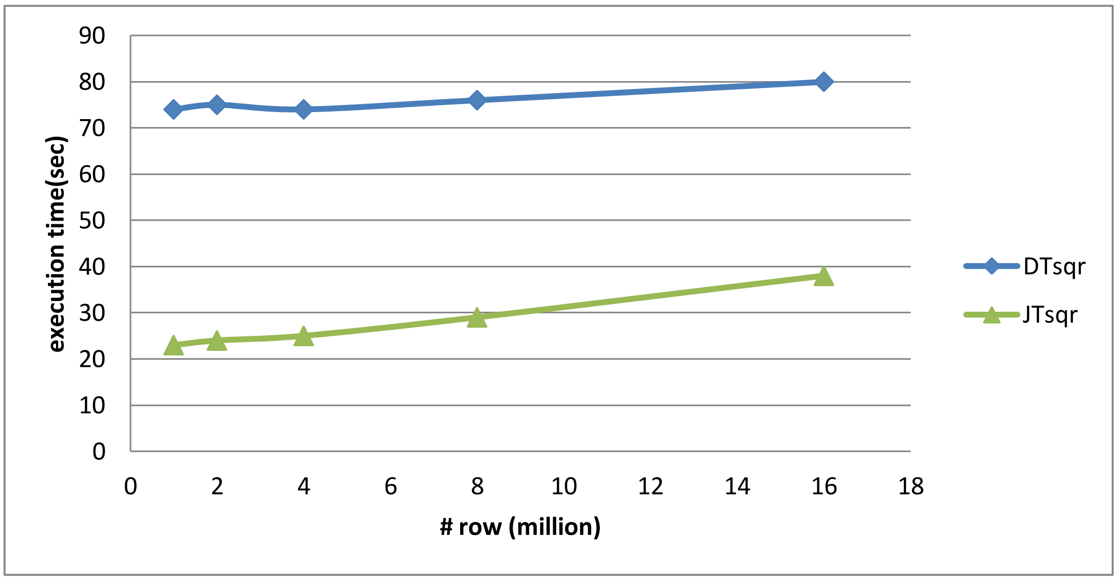

Figure 5 Performance comparison between JTSQR and DTSQR, the column size of matrices is 10, "reduceSchedule" of JTSQR is "1".

There are two reasons why JTSQR can outperform DTSQR. The first is the language differences. JTSQR is implemented in Java and DTSQR is in Python. Since the Hadoop framework is written in Java, and Python code uses the Hadoop Streaming API to pass data between Python Map and Reduce code [2], an overhead will be added to Python MapReduce. The second reason is DTSQR uses two MapReduce iterations to compute QR factorization, which are combined in one MapReduce iteration in JTSQR. That avoids the time of disk I/O and initialization of launching MapReduce task.

3.3 The Impact of Argument "reduceSchedule" on TSQR Performance

The introduction of argument "reduceSchedule" is in the section 2.2. QR Factorization of JTSQR. With this argument, JTSQR could handle bigger data and could avoid out of memory exception of JVM.

In this experiment, we compare the performance of JTSQR using different two values of argument "reduceSchedule": "4,1" and "1". To simulate the cases of extremely large matrices, we generate a bunch if R matrices to be merged. There are two experiments. In Figure 6, the sizes of matrices were fixed to 100 and the number of matrices is varied from one thousand (1,000) to eight thousands (8,000). In Figure 7, the number of matrices is fixed to 25 and the matrix sizes were varied from one hundred (100) to eight hundreds (800).

The result shows when the number of matrices increases or the matrix size becomes large, the program with the argument "reduceSchedule" "4,1" outperforms the one with argument "1". In Figure 6, the program using argument "4,1" is better than the one with "1" when number of matrices exceeds 5,000. In Figure 7, the program with argument "4,1" is better than the one with "1" when matrix size is bigger than 400. It should note that in Figure 6, program that uses argument "1" will cause out of memory when the number of matrices is 8000, and in Figure 7, when the size of matrices is larger than 600, the program with "reds:1" causes out of memory.

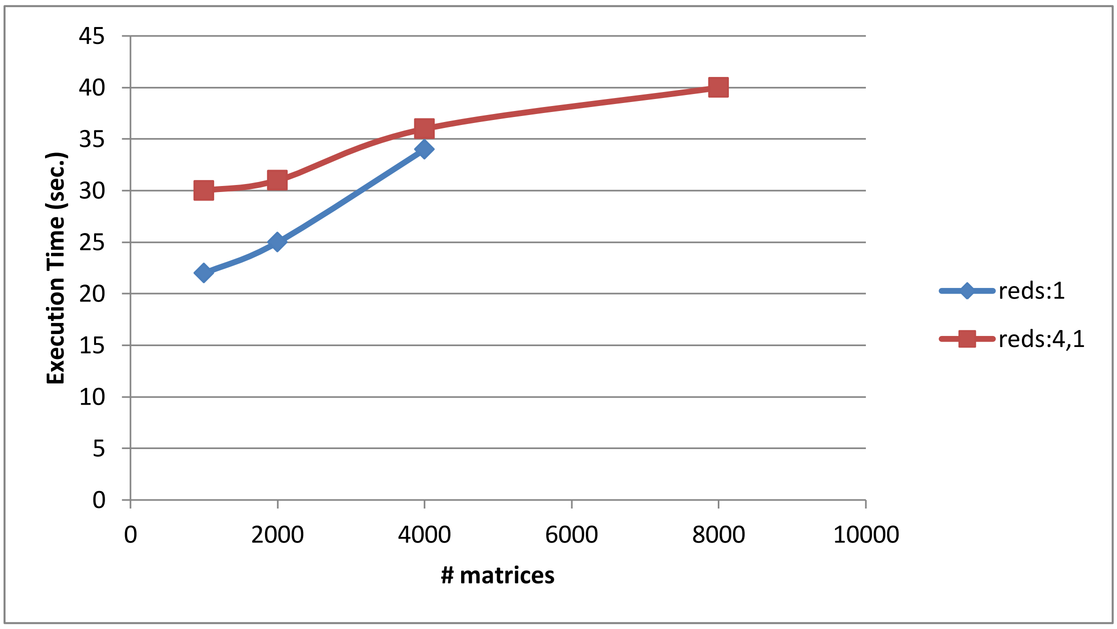

Figure 6 Performance comparison between setup of "reduceSchedule" "1" and "4,1", the matrix size is 100.

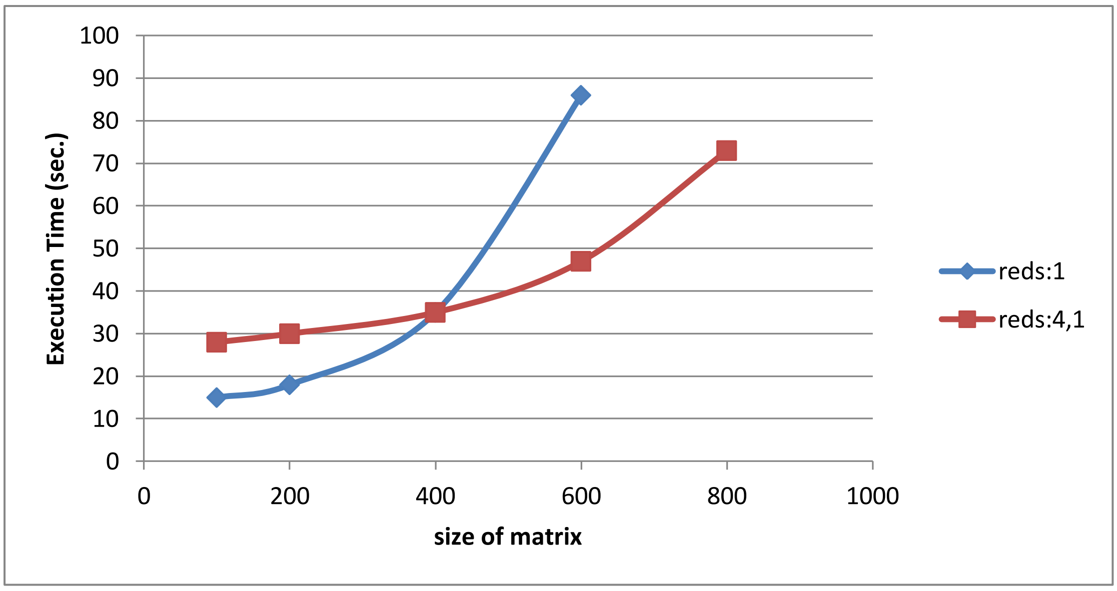

Figure 7 Performance comparison between setup of "reduceSchedule" "1" and "4,1", the number of matrices is 100.

4. References

[1] Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures
[[pdf] http://arxiv.org/abs/1301.1071]

[2] Writing an Hadoop MapReduce Program in Python
[http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/#!]

[3] Matrix-toolkits-java
[https://github.com/fommil/matrix-toolkits-java]