Linear Algebra and its Application

Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures

Manami Das
MDS202423

Maria Paul Thurkadayil
MDS202424

Mohit Singh Sinsniwal
MDS202425

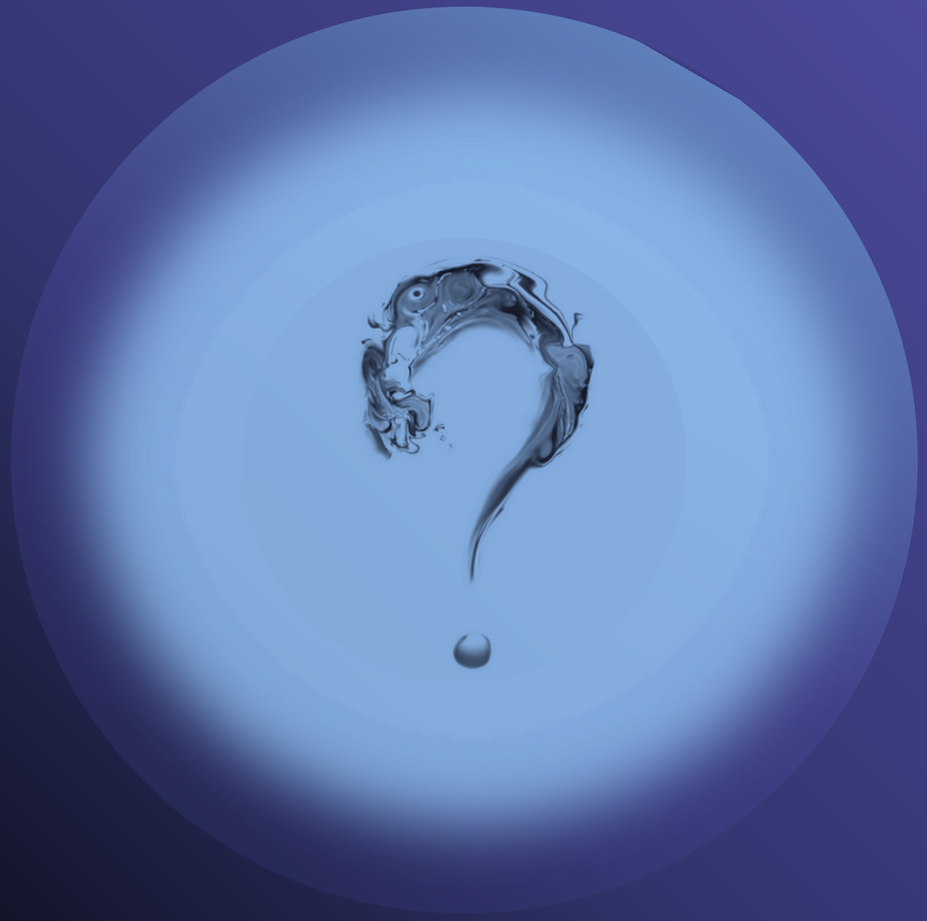
Acknowledgement

This presentation is based on the research paper
“Direct QR Factorizations for Tall-and-Skinny Matrices in MapReduce Architectures”
by Austin R. Benson, David F. Gleich, and James Demmel.
We acknowledge the authors for their significant contributions to the field of distributed linear algebra and high-performance computing. Their work has laid the foundation for scalable and numerically stable factorizations in modern data-intensive applications.
We would also like to thank Prof. Kavita Sutar for her support and guidance throughout the preparation of this presentation.

Content

- Motivation and Problem set
- Background
  - Tall and skinny matrices
  - MapReduce Architectures
  - Success Metrices
  - Previous QR methods and their challanges
  - TSQR (Tall-and-Skinny QR)
  - Direct TSQR
- Algorithm
- Implementation
- Experiment Results
- Summary and Key take-aways
- Conclusion and future work
- Applications and Use Cases
- References

Motivation and Problem set



Motivation

- QR factorization and SVD are core tools in scientific computing and data analysis.
- Many real-world datasets generate tall-and-skinny matrices , especially in large-scale data applications.

Problem

- Existing QR methods in MapReduce rely on numerically unstable techniques, such as computing Q indirectly.
- These approaches may be fast (requiring only 2 data passes) but often compromise accuracy.
- Stable alternatives like Householder QR are computationally expensive and not optimized for MapReduce.
- A solution is needed to balance numerical stability with performance in distributed systems.

Goal of the Paper:

Propose a stable, efficient, and scalable algorithm (Direct TSQR) for computing QR (and SVD) of tall-and-skinny matrices in MapReduce, with minimal data passes.

Why This Paper Matters?

- Proposes a direct QR algorithm (TSQR) that:
  - Avoids the indirect method
  - Computes both Q and R stably and efficiently
  - Works well on nearly terabyte-scale matrices
  - Achieves performance close to theoretical lower bounds on runtime

Background

Tall and Skinny Matrices

A matrix $A \in \mathbb{R}^{m \times n}$ is tall-and-skinny when: $m \gg n$

Common in large-scale data settings, especially when each row is a data point and each column is a feature, like:

- Rows represent data samples (e.g., millions of observations)
- Columns represent features (e.g., tens or hundreds)

Why are they important?

- Appear frequently in machine learning, data analytics, and scientific computing.
- Enable tasks like:
  - Principal Component Analysis (PCA)
  - Linear regression
  - Dimensionality reduction
  - Signal and image processing

Challenges in Distributed Environments

- Despite their simple shape, they can be massive in size (millions of rows).
- Need efficient, numerically stable algorithms that work well in MapReduce or cloud-scale systems.

Key Use Case

- When working with distributed datasets (e.g., Hadoop or Spark), tall-and-skinny matrices are ideal for applying parallel QR factorization, especially with communication-avoiding algorithms like TSQR.

Tall and Skinny Matrices

QR Factorization Overview

- **Goal**: Decompose matrix A into:

$$A=QR$$

  - $Q \in \mathbb{R}^{mxn}$: orthogonal matrix (columns are orthonormal)
  - $R \in \mathbb{R}^{mxn}$: upper triangular matrix

Challenge in MapReduce

- Existing methods in MapReduce efficiently compute only R, and rely on the indirect formula for Q:

$$Q=AR^{-1}$$

- Assumes A is full-rank but is often numerically unstable.
- May require iterative refinement to improve accuracy, which is costly and still error-prone if the matrix is ill-conditioned.
- Can result in large errors in Q^TQ(should ≈ identity matrix)

Limitations of Iterative Refinement

- One workaround is to refine Q by repeating the QR process.
- However:
  - Expensive computationally
  - Still inaccurate if A is badly conditioned
  - Not scalable in MapReduce environments

Quick Overview of MapReduce

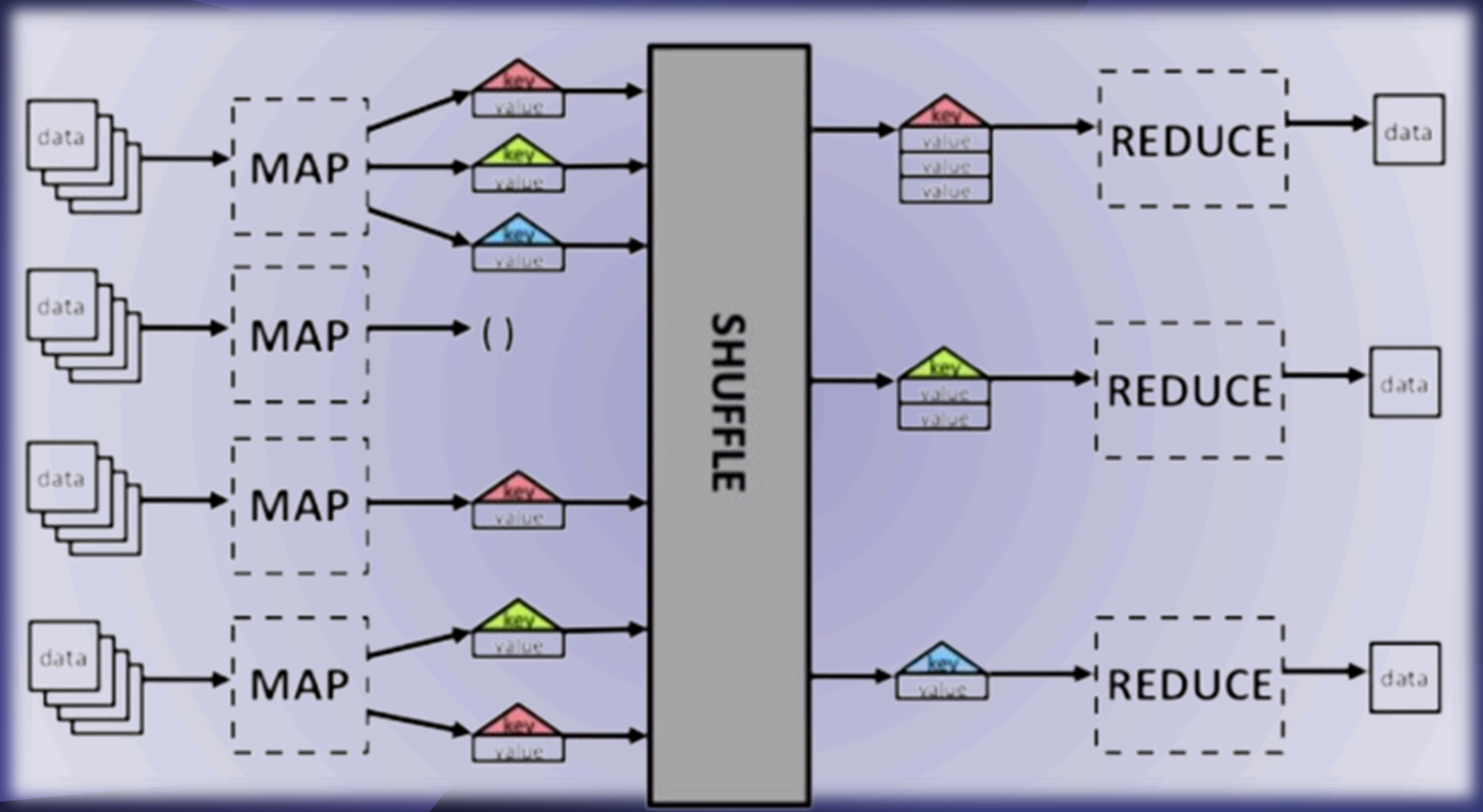
Basic Concept

- MapReduce operates on collections of key-value pairs.
  - In matrix computations:
    - Key = Row identifier
    - Value = Row data (elements of the matrix)
- Each matrix row is processed independently, making it well-suited for distributed environments.

Execution Model

1. Map Phase:
   - Applies a function to each key-value pair
   - Outputs transformed key-value pairs
2. Shuffle Phase:
   - Redistributes data so that all values with the same key are grouped together
3. Reduce Phase:
   - Applies a function to all grouped values
   - Produces final aggregated results

Quick Overview of MapReduce



What Are "2 Data Passes" in MapReduce?

- In the context of MapReduce or any distributed computing system, a "data pass" refers to how many times the system must read through the entire dataset to perform a computation.
- When an algorithm is said to require 2 data passes, it means:

First Pass: The system scans through the full matrix once—typically to compute intermediate results (like local QR factorizations or partial statistics).

Second Pass: It goes through the data again—usually to reconstruct final results (like assembling the full Q matrix or refining the R factor).

- Each pass:

Involves reading the full dataset from storage or memory.

Incurs communication and I/O cost, which can be expensive in distributed systems.

Why It Matters:

- Fewer data passes = better performance, especially on massive datasets.
- MapReduce environments are I/O-heavy, so minimizing passes improves efficiency.

Quick Overview of MapReduce

Storage & Performance

- Hadoop, the most common MapReduce platform, stores both data and intermediate results on disk.
- This makes MapReduce slower than in-memory systems (e.g., MPI), but much more scalable and fault-tolerant.

Why Use MapReduce?

- Many massive datasets already reside in MapReduce clusters, avoiding costly data transfers.
- Offers:
  - Transparent fault tolerance (e.g., automatic recovery from node failures)
  - Simplified I/O management
  - High adaptability and ease of implementation

Implementation Notes

- Used Hadoop Streaming with the Dumbo (Python) interface
  - ~70 lines of Python vs. ~600 lines in C++ code
  - Python was easier to maintain; C++ only gave ~2× speedup
- Other MapReduce platforms include:
  - Twister, Phoenix++, LEMOMR, MRMPI
  - Some offer in-memory speed but lack Hadoop's fault tolerance

Success Metrics

The two success metrics used to evaluate the algorithms in the paper are:

1. Speed (Performance)

- Evaluated using a performance model tailored to the MapReduce cluster.
- Model fits just two parameters and predicts runtime within a factor of two of actual results.
- Results shown in Section V-B demonstrate:
  - Direct TSQR is significantly faster than classic stable methods (e.g., Householder QR).
  - Achieves near-optimal performance relative to theoretical lower bounds.

2. Stability (Numerical Accuracy)

We use two standard metrics to evaluate the quality of the QR decomposition:

Metric	Measures	Ideal Value
$\frac{\|A-QR\|_2}{\|R\|_2}$	**Accuracy** of the decomposition	$O(\epsilon)$
$\|Q^TQ - I\|_2$	**Orthogonality** of Q	$O(\epsilon)$

Where ϵ(epsilon) is machine precision in implementation reconstruction error is found which is $||A - QR||_F$

Challenges with Traditional QR in MapReduce

1. Cholesky QR (Indirect QR factorisation)

- Approach:
  - Computes A^TA, then performs Cholesky factorization:

$$A^TA=LL^T \text{ (here L is lower triangular)and } A^TA=R^TR \text{ (for QR factorisation)} \Rightarrow R=L^T$$

- Simple and parallelizable, but:
  - Highly sensitive to conditioning:

$$\kappa(A^TA)=\kappa(A)^2$$

  - Leads to significant loss of numerical stability for ill-conditioned matrices.
- Result: Unstable Q factor due to amplified round-off errors.
- Limited parallelism in the reduction phase: at most n reduce tasks due to dimensionality.

2. Householder QR in MapReduce (Direct QR factorisation)

- Approach: Classic stable algorithm using reflectors.
- Problem: Not MapReduce-friendly:
  - Requires 2n passes over data for n columns.
  - Frequent disk rewrites slow down performance.
- Incompatible with HDFS row-wise layout and Hadoop's I/O model.
- Worse scalability as n grows.

Introduction to TSQR (Tall-and-Skinny QR)

Why TSQR?

- Problem with Cholesky QR:
- Condition(A^TA)=(Condition(A))^2

→ High sensitivity to numerical errors

→ Leads to inaccurate R in finite precision

TSQR: A More Stable Alternative

- Developed by Demmel et al., adapted by Constantine & Gleich for MapReduce.
- Known to be numerically stable.
- Particularly well-suited for tall-and-skinny matrices (i.e., $m \gg n$).

Indirect TSQR is designed to compute R stably in a distributed setting using hierarchical QR:

$A=QR$, but only R is computed directly

- Q is not formed explicitly in the reduction step.

Introduction to TSQR (Tall-and-Skinny QR)

Step-by-Step Flow :

1. Partition A into row blocks across map tasks:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \cdots \\ A_p \end{bmatrix}$$

2. Each mapper computes a local QR:

$$A_i = Q_i R_i$$

3. Stack the R_i's and compute their QR:

$$\begin{bmatrix} R_1 \\ R_2 \\ \cdots \\ R_p \end{bmatrix} = \widetilde{Q}\widetilde{R}$$

4. Final factor:

$$A = \underbrace{\begin{bmatrix} Q_1 \\ Q_2 \\ \cdots \\ Q_p \end{bmatrix} \widetilde{Q}}_{\text{Not computed}} \cdot \widetilde{R}$$

> Only $\widetilde{R}$ is returned

> To get Q, use $Q = AR^{-1}$ indirectly

Introduction to TSQR (Tall-and-Skinny QR)

Stability Analysis

- Numerical stability of R is high due to hierarchical QR.
- But:

$Q=AR^{-1}$ is not backward stable

Especially problematic for ill-conditioned matrices

Orthogonality error:

$$||Q^TQ - I||_2 \gg \epsilon$$

Iterative Refinement

- Often used to improve Q:
  a. Compute $Q=AR^{-1}$
  b. Apply another QR factorization on Q to correct it
- Works only when matrix A is well-conditioned

About Computing $Q=AR^{-1}$:

Used in Cholesky QR and Indirect TSQR when Q isn't formed directly.

Pros:

Fast and parallelizable

Cons:

Numerically unstable, especially for ill-conditioned matrices

Not backward stable, i.e., even refinement might not recover good Q

Why Direct TSQR?

- Avoids the need for computing AR^{-1}
- Computes Q and R directly
- Ensures numerical stability and high performance

Direct TSQR: The Proposed Method

Goal:

- Compute a numerically stable QR decomposition in a MapReduce setting using 3 passes over the data — avoids the instability of Cholesky QR and the indirect Q computation of TSQR.

3-Step Workflow

- Step 1: Local QR (Map Tasks Only)

  Partition matrix A row-wise across mappers:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}$$

  Each mapper computes:

$$A_i = Q_i R_i$$

  Emit Q_i and R_i to separate files.

- Step 2: Global QR on R (Single Reduce Task)

  Collect R_1, R_2, R_3, R_4 into a stacked matrix:

$$\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} = Q^{(2)} \widetilde{R}$$

  Emit sections of $Q^{(2)}$ corresponding to each R_i .

Direct TSQR: The Proposed Method

- Step 3: Final Q Construction (Map Tasks Again)

  Multiply local Qs with their corresponding global Q factors:

$$Q = \begin{bmatrix} Q_1 Q_1^{(2)} \\ Q_2 Q_2^{(2)} \\ Q_3 Q_3^{(2)} \\ Q_4 Q_4^{(2)} \end{bmatrix}$$

  Final result:

$$A = Q\tilde{R}$$

Advantages

- Numerically stable for both Q and R
- Simple and parallel-friendly
- Avoids disk-heavy iterative updates (like Householder QR)
- Adaptable to compute SVD with one extra step

Extension: SVD from Direct TSQR

- After Step 2, compute: $\tilde{R} = U\Sigma V^T$
- Final result: $A = (QU)\Sigma V^T$

Can skip storing Q on disk if only QU or Σ needed

	Indirect TSQR	Direct TSQR
Objective	Compute R directly, infer $Q = AR^{-1}$	Compute **both** Q and R **directly**
Q Computation	Indirect (via matrix inverse, unstable)	Direct (via multiplication of local and tree-level Q)
Stability	❌ Poor Q orthogonality if A is ill-conditioned	✅ Numerically stable for both Q and R
Refinement Needed	Often (to improve Q)	Not required
Map Tasks	Local QR: $A_i = Q_i R_i$	Same
Reduce Tasks	QR on R_i stack to get $\tilde{R}$	Also uses $\tilde{Q}$ to reconstruct global Q
Final Output	Only R, optionally $Q = AR^{-1}$	Full QR: $A = QR$
Use Case Fit	When only R is needed (e.g. SVD, PCA)	When full QR is required (e.g. regression, Gram-Schmidt)

Algorithm

Cholesky QR

Algorithm 1 Compute $A^T A$ in MapReduce

```
function MAP(key k, val a)
    for i, row in enumerate(a^T a) do
        emit(i, row)
    end for
end function

function REDUCE(key k, ⟨ vals v_j^k ⟩)
    emit(k, sum(⟨v_j^k⟩))
end function
```

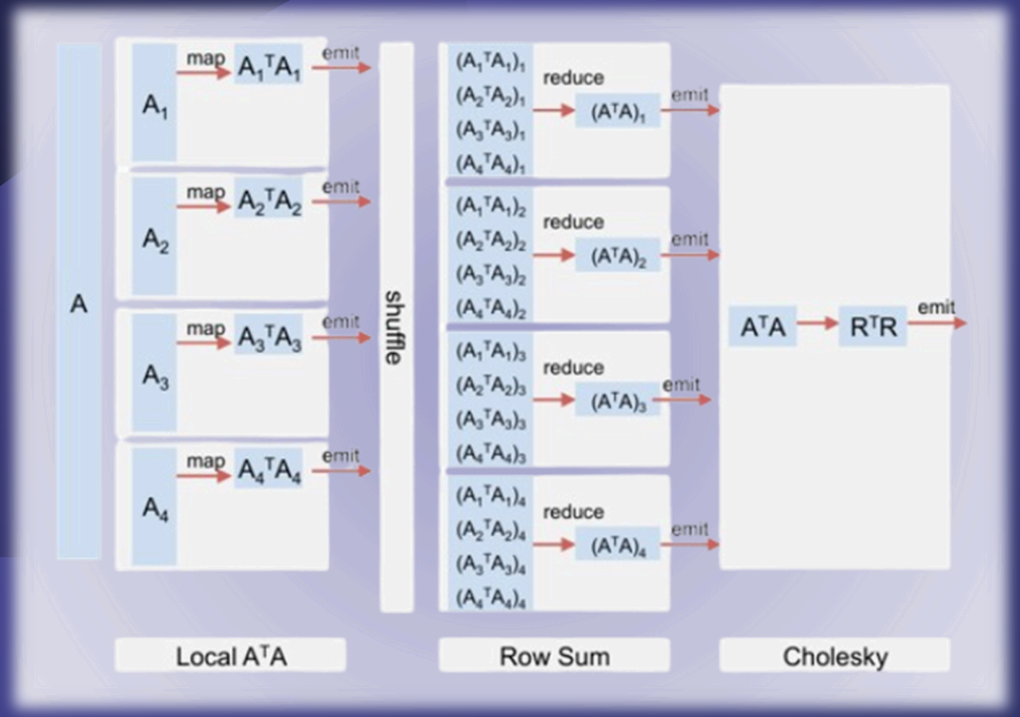


Figure1. MapReduce CholeskyQR computation for a matrix A with 4 columns

Map Phase:

Input: key = row index, value = row vector of A

Each map task:

Forms a local matrix A_p

Computes $A^T_p A_p$ (small n×n matrix)

Emits each row of $A^T_p A_p$ as (key=i, value=row vector)

Reduce Phase:

For each key i (row index in A^TA):

Collect all row vectors row_i^j from mappers

Sum them row wise: $A^TA = \sum_p A^T_p A_p$

Scalability & Design Notes:

- Output size is n rows (each row is a key).
- Simple and parallelizable for tall-and-skinny matrices
- Works well when n (number of columns) is small
- Up to n reduce tasks can be used.
- Cholesky factorization of the final A^TA is fast . It can be assembled by summing mapper outputs
- Scalability limited by size of n×n matrix

MapReduce Flow: Compute A^TA

Indirect TSQR

Algorithm2 Indirect TSQR in MapReduce

function map(key k, val A_k)
// A_k : Local chunk of the tall matrix A
$[Q_k, R_k] = qr(A_k)$ // Local QR factorization
emit(k, R_k) // Emit R_k for final QR
end function

function reduce(key k, values $[R_1, R_2, ..., R_p]$)
// Stack all R_k vertically to form R_{stack}
R_{stack} = stack_vertically($[R_1, R_2, ..., R_p]$)
$[\_, R_{tilde}] = qr(R_{stack})$ // Final QR
emit("R_final", R_{tilde}) // Final upper triangular R
end function

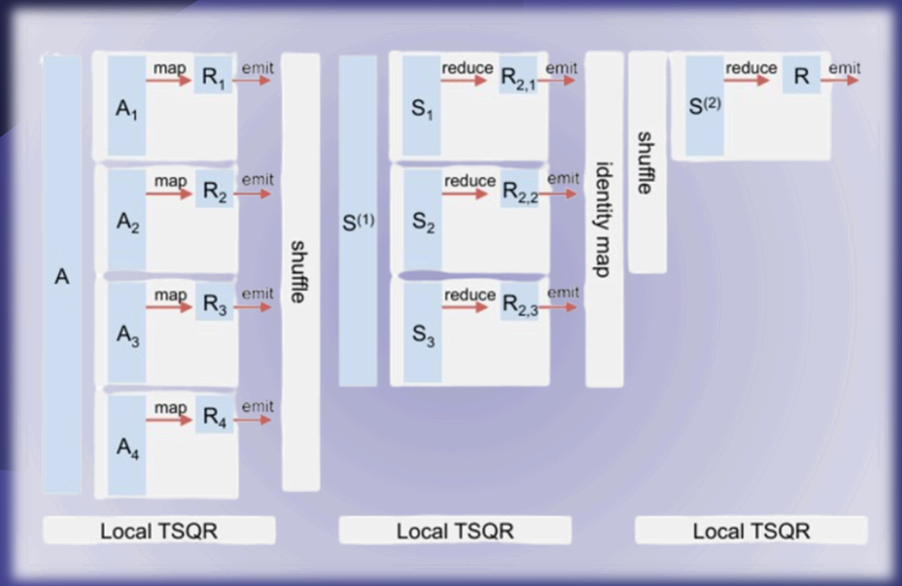


Figure2.MapReduce TSQR computation .$S^{(1)}$ is the matrix consisting of the rows of the R_i factors stacked on top of each other, i=1,2,3,4. Similarly, $S^{(2)}$ is the matrix consisting of the rows of the $R_{2,j}$ factors stacked on top of each other, j=1,2,3.

Map Phase:

Input: key = row index, value = row vector of A

Each map task:

Takes a local chunk A_k

Computes a local QR factorization: $A_k = Q_k R_k$

Emits only R_k — the upper triangular matrix — to the next stage.

Reduce Phase:

A reducer takes all R_k matrices:

Stacks them vertically into a matrix R_{stack}

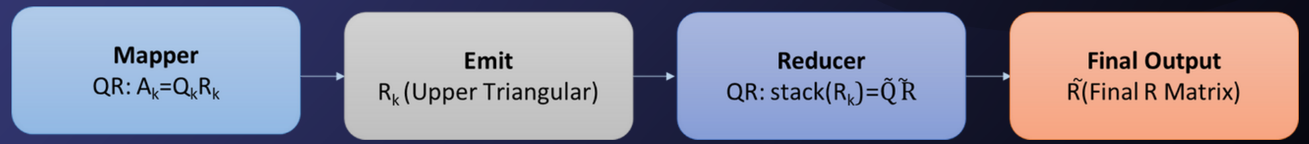
Computes another QR:

$R_{stack} = \tilde{Q}\,\tilde{R}$

Emits $\tilde{R}$ as the final R of the QR factorization of A.

Scalability & Design Notes:

- Parallelism: Each mapper runs independently → Excellent parallel scalability.
- Multi-stage MapReduce: Using an additional MapReduce iteration to structure a binary tree reduction further improves performance on large clusters.
- No need for full matrix aggregation: Unlike Cholesky QR, there's no need to compute or store the full A^TA or Q.
- Scalability limited by size of n×n matrix.



MapReduce Flow: Indirect TSQR

Why Don’t We Emit Q?

- Computing the full matrix Q would require keeping track of intermediate Qk and multiplying them by $\tilde{Q}$.
- This increases:
  - I/O cost (writing and reading large Q matrices),
  - Memory usage, and
  - Network communication in MapReduce.
- Instead, we can reconstruct Q later using:

$$Q = A \cdot \tilde{R}^{-1}$$

- This is why it's called "Indirect TSQR".

Algorithm3 Recover $Q = AR^{-1}$ in MapReduce

```
function map(key k, val A_k)
        // A_k : Local chunk of the tall matrix A
         load R_final                  // Broadcast from reducer
         R_inv = inverse(R_final)
         Q_k = A_k * R_inv             // Matrix multiply
         emit(k, Q_k)
         end function
```

Scalability & Design Notes:

- R^{-1} is small and can be broadcast efficiently
- Fully parallelizable: each mapper computes its part independently
- Output size = m rows total (each row in Q)
- Avoids large intermediate storage (does not require storing all Q_ks beforehand)

Recover Q

Map Phase:

Input: key = row index, value = row vector of A

Each map task:

Takes a local chunk A_k

Loads the global R_{final}(from previous reduce step)

Computes R^{-1} (cheap: upper triangular, small n×n)

Computes: $Q_k = A_k \cdot R^{-1}$

Emits (key=k, value=Q_k)

Reduce Phase:

Not required.

Each map task emits its portion of Q.

The final Q is the vertical stack of all Q_k blocks.

Householder QR

Algorithm4 Householder QR in MapReduce

```
for j = 1 to n do:

   // ----- Step 1: Compute Householder vector -----
   function map1(key i , row Ai)
      emit(0, Ai[j:])                          // Send part of row to single reducer
   end function

   function reduce1(key 0, values [rows])
      compute norm σ = ||A[j:m, j]||
      compute Householder vector v
      broadcast v to all nodes
   end function

   // ----- Step 2: Apply Householder Transformation -----
   function map2(key i, row Ai)
      load v                                   // Broadcast from reduce1
      Ai[j:] = Ai[j:] - 2 * v * (vT*  Ai[j:])
      emit(i , Ai)                             // Updated row
   end function

end for
```

Map Phase 1: Compute Householder vector

Input: key = row index, value = row vector of A_i

Each map task:

Takes a local row A_i

Emits trailing part of row $A_i[j:]$ to reducer for computing the Householder vector

Reduce Phase 1: Form Householder Vector

Reducer:

- Receives partial columns $A_i[j:]$ from all mappers
- Computes:
  - Norm $\sigma = ||A[j:m, j]||$
  - Forms Householder reflector vector v
- Broadcasts v to all mappers

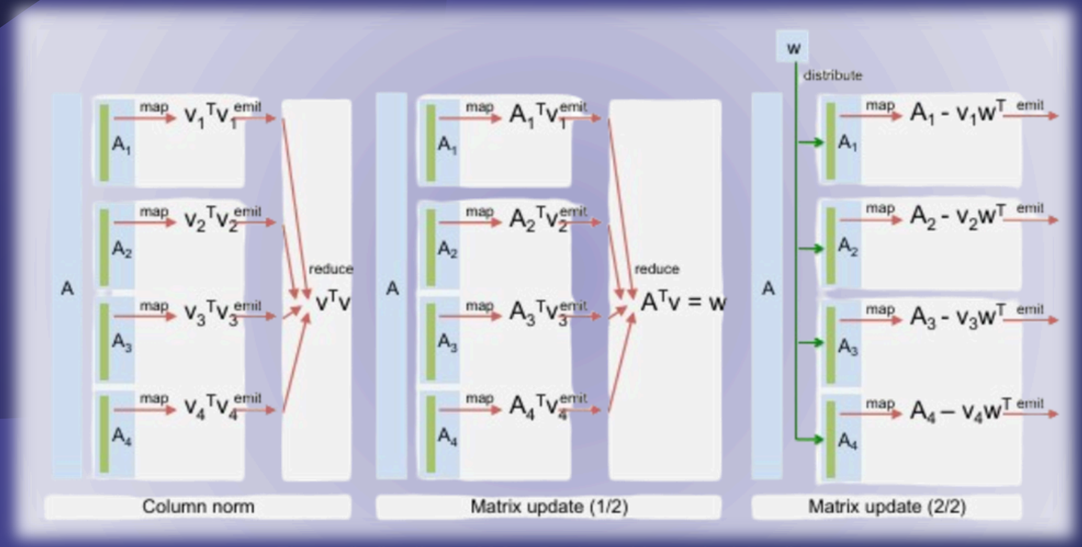


Figure3. Outline of MapReduce Householder QR

Map Phase 2: Compute Householder vector

Input: key = row index, value = row vector A_i

Each map task:

Loads Householder vector v

Applies transformation:

$$A_i[j:] = A_i[j:] - 2 \cdot v \cdot (v^T A_i[j:])$$

Emits updated row A_i

Loop repeats for each column j = 1 to n

Output:

Final matrix R (upper triangular)

Householder QR

Repeat

- Repeat Steps 1 and 2 for each column j=1 to n
- Total of 2n MapReduce passes for an m×n matrix

Scalability & Design Notes:

- Multiple passes: Requires 2n MapReduce jobs for n columns.
- High I/O cost: After each iteration, the full matrix A is rewritten.
- Sequential Nature: Each step depends on the previous; no parallelism across columns.
- Numerical Accuracy: Very high due to Householder stability.
- BLAS-2 style computation: Less efficient than BLAS-3 approaches used in ScaLAPACK.
- Disk-heavy: Matrix rewriting leads to performance bottlenecks in large systems.
- Not well-suited for HDFS (row-wise layout) → Cannot leverage fast column access.
- Better alternatives for MapReduce environments: Indirect TSQR or Cholesky QR.
- Scalability Limited: Works well for small nnn, but suffers for wider matrices due to iterative structure and high I/O cost.

Direct TSQR

Algorithm5 Direct TSQR in MapReduce

```
// Step 1: Local QR on chunks of A
function map1(key k, val A_k ):
    // A_k is a local chunk of matrix A (m × n)
    [Q_k, R_k] = qr (A_k)                    // Local QR
    emit_to_file("Q_chunk", (k, Q_k))     // Save local Q part (e.g., Feather or HDFS file)
    emit_to_file(" R_chunk ", (k, R_k))   // Save local R for next step
end function
// Step 2: Global QR on stacked R matrices
function reduce(key dummy, values [R_1, R_2, ..., R_p]):
    R_stack= stack_vertically ([R_1, R_2, ..., R_p ])  // Stack local R_k matrices
    [Q2, R_final] = qr(R_stack)                        // Global QR
    emit_to_file(" Q2_chunk ", Q2)                     // Save Q2 split by blocks
    emit(" R_final", R_final)                          // Output final R matrix
end function
// Step 3: Multiply Q1 and Q2 to get final Q
function map2(key k, val Q1_k):
    Q2 = load_file("Q2_chunk ")            // Load Q2 portion for block k
    Q2_k = get_block(Q2, k)                // Extract matching Q2 block
    Q_final_k = Q1_k × Q2_k                // Final Q block
    emit(k, Q_final_k )                    // Emit final rows of Q
end function
```

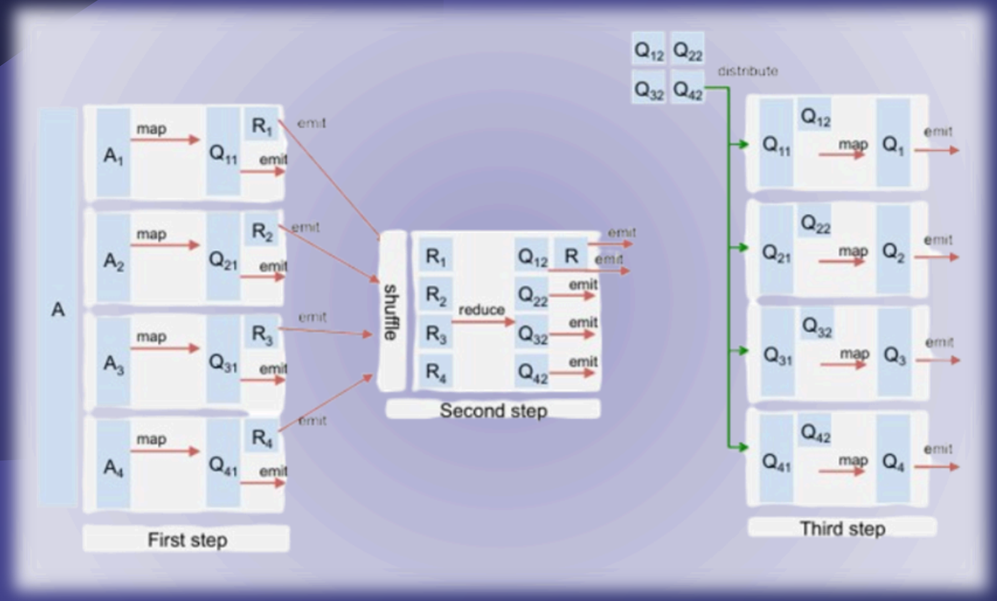


Figure4. Direct MapReduce computation of Q and R .

Map Phase 1: Local QR on Chunks of A

Input: key = row index, value = A_k (local chunk of A)

Each map task:

Performs QR factorization: $A_k = Q_k R_k$

Emits:

Local orthogonal matrix Q_k to "Q_{chunk}" file

Local upper-triangular matrix R_k to "R_{chunk}" file

Reduce Phase 1: Global QR on R Matrices

Reducer:

Input: All R_k matrices from Step 1

The reducer:

Stacks the matrices vertically to form R_{stack}

Performs QR: $R_{stack} = Q_2 \times R_{final}$

Emits:

Q_2 → saved to "Q2_chunk" (row-split blocks)

R_{final} → final R of the full matrix A

Map Phase 2: Compute Final Q

- **Input:** (k, $Q2_k$) from "$Q2_{chunk}$"

Each map task:

Loads matching part Q2_k from "Q2_chunk"

Computes: $Q_{k\_final} = Q1_k \times Q2_k$

Emits updated row A_i

Direct TSQR

Key Idea

Direct TSQR avoids the iterative nature of Householder QR and instead:

Performs local QR → then global QR on the R's → then multiplies Q_1 and Q_2.

This gives:

$A = Q \times R_{final}$, where

$Q = Q_1 \times Q_2$ and R_{final} is the final upper-triangular matrix.

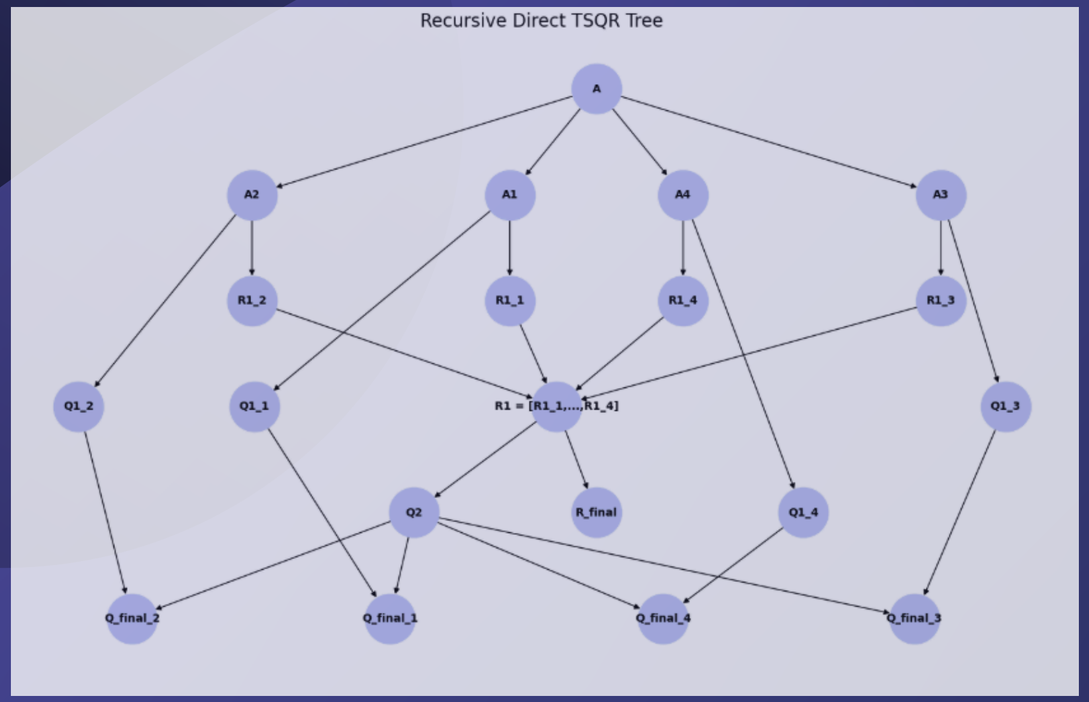
Scalability & Design Notes

- Parallelism: Step 1 and Step 3 map phases scale perfectly — each chunk is processed independently.
- Efficient use of storage: Q and R matrices stored separately in files (e.g., Feather, HDFS).
- No repeated passes: Unlike Householder QR, this only requires 2 passes over the data.
- Numerically stable: Leverages QR on smaller R matrices rather than A^TA as in Cholesky.
- Limitation: Final multiplication step requires all tasks to access portions of Q_2.

Extending Direct TSQR to a recursive algorithm

Algorithm6 Recursive extension of direct method

```
function DirectTSQR(matrixA)
        Q1,R1=FirstStep(A)
        if R1 is too big then Assign keys to rows of R1
                Q2=DirectTSQR(R1)
        else
                Q2=SecondStep(R1)
        endif
        Q=ThirdStep(Q1,Q2)
        return Q
end function
```



Recursive Direct TSQR: Overcoming Bottlenecks

Motivation

- In Direct TSQR, all local R_k blocks (from step 1) are stacked and QR-decomposed on a single reducer.
- This becomes a scalability bottleneck as the matrix becomes "fatter" (more columns or many partitions).
- **Solution**: Use a recursive strategy that hierarchically performs QR factorizations

Core Idea

Instead of aggregating all R_k at once, we:

1. Apply Direct TSQR to matrix A to get:
   - Local Q blocks → Q1
   - Stacked R blocks → R1
2. Check if R1 is still too large to QR in memory.
3. If yes → recursively call DirectTSQR on R1
4. Else → proceed with normal second step QR
5. Final $Q=Q1 \cdot Q2$, like before

Scalability & Design Notes

- Recursive reduction tree avoids large memory loads on any one reducer
- Works efficiently for both tall-and-skinny and fat matrices
- Adds log-depth recursion, similar to binary reduction trees

Implementation

Implementation Overview

- Dask used for distributed computation.
- Matrix is partitioned → local QR → global reduce.
- Implemented:
  - cholesky_qr(A)
  - householder_qr(A)
  - indirect_tsqr(A)
  - direct_tsqr(A)
  - cholesky_qr(A)

Evaluation Metrics

- Reconstruction Error $\|A - QR\|$
- Orthogonality $\|Q^TQ - I\|$

```
def reconstruction_error(A, Q, R):
    return norm(A - Q @ R) / norm(R)

def orthogonality_error(Q)
    return norm(Q.T @ Q - I)
```

Experiment Results

QR Error Analysis on Diverse Matrices

- getErrors(A_bad)
- Ill-conditioned Matrix (Not Positive Definite)
- Cholesky QR fails due to non-PD input.
- TSQR and Householder maintain high accuracy.

```
getErrors(A_bad)
```

```
🎯 QR Error Metrics

⚠️ Error in Cholesky QR: Matrix is not positive definite
Method                    Reconstruction Error      Orthogonality Error
Cholesky QR                 ⚠️ Error                ⚠️ Error
Indirect TSQR               5.58e-11                3.17e+00
Direct TSQR                 6.78e-16                1.06e-15
Householder QR              3.33e-16                4.75e-16
```

- getErrors(A_tall_skinny)
- Well-behaved Matrix
- Full column rank, well-scaled, non-collinear.
- All methods show excellent reconstruction and orthogonality.

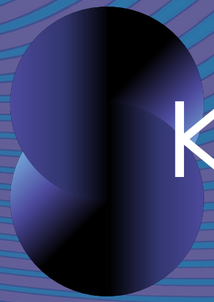
```
getErrors(A_tall_skinny)
```

```
🎯 QR Error Metrics

Method                    Reconstruction Error      Orthogonality Error
Cholesky QR                 1.37e-16                4.59e-15
Indirect TSQR               2.03e-16                2.69e-16
Direct TSQR                 2.03e-16                2.69e-16
Householder QR              2.03e-16                2.69e-16
```

- getErrors(A_high_kappa)
- Il-conditioned Matrix (κ ≈ 1e8)
- High condition number tests numerical stability.
- Cholesky QR shows significant orthogonality loss.
- TSQR and Householder remain numerically stable.

```
getErrors(A_high_kappa)
```

```
🎯 QR Error Metrics

Method                    Reconstruction Error      Orthogonality Error
Cholesky QR                 1.26e-16                2.53e-01
Indirect TSQR               1.57e-16                8.47e-09
Direct TSQR                 3.42e-16                7.33e-16
Householder QR              1.28e-16                3.37e-16
```



Key take-aways

Key Takeaways

Cholesky QR: fast but numerically weaker.

Householder QR (on full matrix):
Is sequential and expensive → not scalable.

Indirect TSQR:

- does not parallelize Q reconstruction → bottleneck if Q is large.

Direct TSQR: achieve high accuracy and parallelism.

Conclusion

Stability Experiment

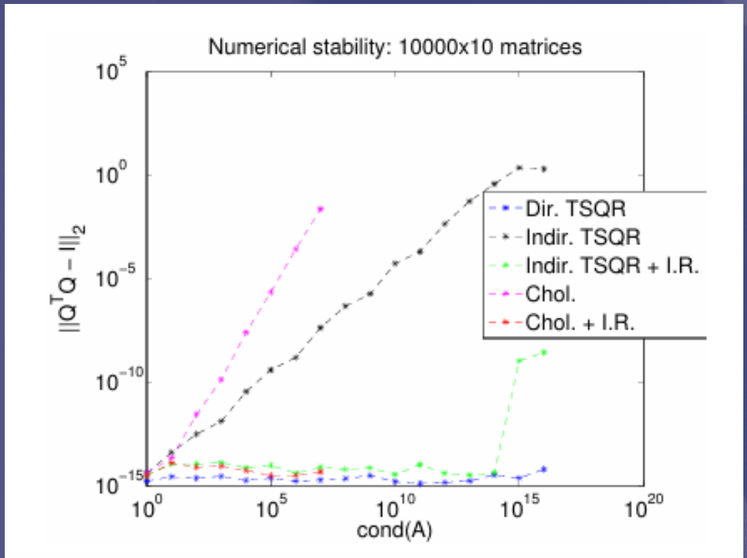


Figure6. Stability measurements for each algorithm for matrices
of varying condition number
Ref: fig 6 in paper page 7

Motivation:

- A major motivation for using the Direct TSQR method is numerical stability.
- Numerical accuracy of QR factorization is critical for ill-conditioned matrices.
- We compare Direct TSQR, Cholesky QR, and Indirect TSQR.

Key Metric:

- Use $||Q^TQ - I||_2$ to measure loss of orthogonality in the computed Q.

Experiment Summary:

- Input: Tall-and-skinny matrices with increasing condition number.
- Output: Accuracy of Q from each method.

Observations (Fig. 6 Overview):

Method	Accuracy Trend
Direct TSQR	~10^{-15} (Stable across all condition numbers)
Indirect TSQR + Refinement	Degrades after condition number ≥ 10^{16}
Cholesky QR	Fails for condition number ≥ 10^{8}
Cholesky QR + Refinement	Better, but fails after condition number ≥ 10^{12}
Indirect TSQR (No Refinement)	High error, scales with condition number

Conclusion

Why Direct TSQR?

- Numerically stable: Guarantees orthogonal Q matrix
- Robust performance: Typically no more than 2× slower than the fastest (but unstable) methods
- Outperforms simpler methods in both accuracy and speed
- Significantly faster than Householder QR in MapReduce

Practical Benefits

- Scales efficiently with MapReduce.
- Works well with tall-and-skinny matrices.
- Orders of magnitude less disk I/O than Householder QR.
- All reference source code is open-source and available at:
- github.com/arbenson/mrtsqr

Applications and Use Cases

Real-World Scenarios for TSQR

- Distributed Linear Algebra:
  - Scalable QR factorizations for large tall-and-skinny matrices
- Machine Learning Pipelines:
  - QR used in least squares, PCA, and feature orthogonalization
  - Preprocessing step for:
    - Low-rank matrix approximation
    - Topic modeling (e.g., LSA)
    - Recommendation systems
  - Scales well with tall-and-skinny data (e.g., user-item matrices)
- Scientific Computing:
  - Handles massive matrices in climate modeling, genomics, etc.
  - Solving overdetermined linear systems using least squares
  - Stable decomposition of large experimental data sets
  - Used in PCA (Principal Component Analysis) for dimensionality reduction
- Cloud & Big Data Systems:
  - Runs seamlessly on Hadoop, Spark, and cloud clusters
  - Real-time log analytics with millions of rows, but few features
  - Text mining pipelines in distributed NLP tasks
  - Feature extraction on distributed datasets
  - Integrated with data lakes for batch processing of large tabular data
- Data Compression & Dimensionality Reduction:
  - QR + SVD used for efficient low-rank approximations

Thank You