Benchmarking SQL and NumPy for Matrix Multiplication

Tristan Hodgson

December 18, 2025

Abstract

We benchmark matrix multiplication performed entirely inside PostgreSQL against a conventional Python workflow using NumPy. We additionally model a slow 10 Mbps connection by adding a transfer-time penalty to the Python workflow. For dense matrices, NumPy consistently outperforms the SQL approach by a wide margin, and the gap grows with matrix size. For sufficiently sparse matrices, the SQL approach becomes competitive and can be faster, as the database can filter out a large fraction of unnecessary multiplications while also eliminating client-side data transfer. These results suggest in-database multiplication can be advantageous for sparse workloads or constrained network conditions, while dense multiplication remains better suited to NumPy.

1 Introduction

Matrices are very common in modern mathematics and due to this ubiquity and general usefulness they are very frequently stored in relational databases such as Postgres. One of the most common operations on matrices is multiplication. A first approach to multiplying matrices stored in a database would be to pull the data into a programming language such as Python where we can use highly optimised libraries to multiply the matrices quickly, and then write the result back into the database.

In this project, we develop an alternative: multiplying the matrices entirely inside of SQL. We then proceed to benchmark this SQL only approach against the more traditional NumPy/Python approach. We anticipate that this might be faster for some matrices since we don't have to export all of the data from SQL into Python and then back again. We also investigate how slow internet speeds could affect these results.

2 Methodology

Matrix storage We store each matrix in a new table. Each row of the table has three columns $(i : INT, j : INT, value : DOUBLE)$. For simplicity, we store *all* entries of the matrix explicitly, including zeros (i.e. there are no implied 0s). This encoding is convenient but means the amount of data stored and transferred is independent of how sparse the matrix is.

Matrix generation For each benchmark setting we generate matrices in NumPy by sampling i.i.d. entries from a uniform distribution on $[0, 1)$. We then apply an elementwise Bernoulli mask with parameter d (the *nonzero proportion*): each entry is kept with probability d and set to 0 otherwise. All entries, including zeros created by masking, are inserted into the database under the storage scheme above. For a fixed d, we generate two matrices at the maximum size used in the experiment and obtain smaller $n \times n$ matrices by truncating to the top-left $n \times n$ block.

SQL Multiplication Matrix multiplication can naturally be written as a summation $(AB)_{ij} = \sum_k A_{ik}B_{kj}$. It is this that we use to calculate our product, with the query below being the natural translation.

```sql
CREATE TABLE C AS
    SELECT A.i, B.j, SUM(A.value * B.value) AS value
    FROM A JOIN B ON A.j = B.i
    WHERE A.value != 0 and B.value != 0
    GROUP BY A.i, B.j;
```

This translation allows us to take advantage of the parallelisation of SQL with little to no work on our part.

Python Multiplication For multiplying in Python we use NumPy's built in function. The workflow then becomes: download the matrices into NumPy arrays, compute the product, upload the new matrix to the SQL server.

Simulated network delay In many scenarios, the SQL server and the client device will not be on the same physical device (e.g. if we used a cloud database provider such as Amazon RDS for PostgreSQL), in this case we must move data across the internet. We also wish to investigate how this may change the relative speed of our approaches.

A slow internet speed of 10Mbps was selected to simulate common scenarios with low internet speed such as a phone on mobile data or a device in an otherwise low connectivity environment (e.g. field work). This speed also has the advantage of making the result more pronounced meaning we can more easily analyse the effect of slow internet speed.

We assume that we begin with the matrices already in the database and want to end up with the product also in the database. As such the Python approach is the only one that needs a delay as it downloads A and B and then uploads (AB). Under our storage scheme we transfer all values regardless of sparsity, so the transferred payload is approximately the values of A, B, and (AB), i.e. $\approx nm + mp + np$ floating point numbers. Approximating each stored value as a 64-bit double, the simulated transfer time is

$$t_{\text{net}} \approx \frac{64(nm + mp + np)}{S \cdot 10^6},$$

where S is the link speed in Mbps. For convenience, in actuality, we also transfer the position (i, j) of each value; we ignore this additional overhead (as well as protocol/serialization overheads) in the delay model.

Benchmarking We benchmark square matrices with $n \in \{10, 100, 300, 500\}$ across several nonzero proportions d. We plot $\Delta t = t_{\text{Python}} - t_{\text{SQL}}$ averaged across three runs. As such $\Delta t > 0$ means SQL is faster, and $\Delta t < 0$ means Python is faster.

3 Results and Discussion

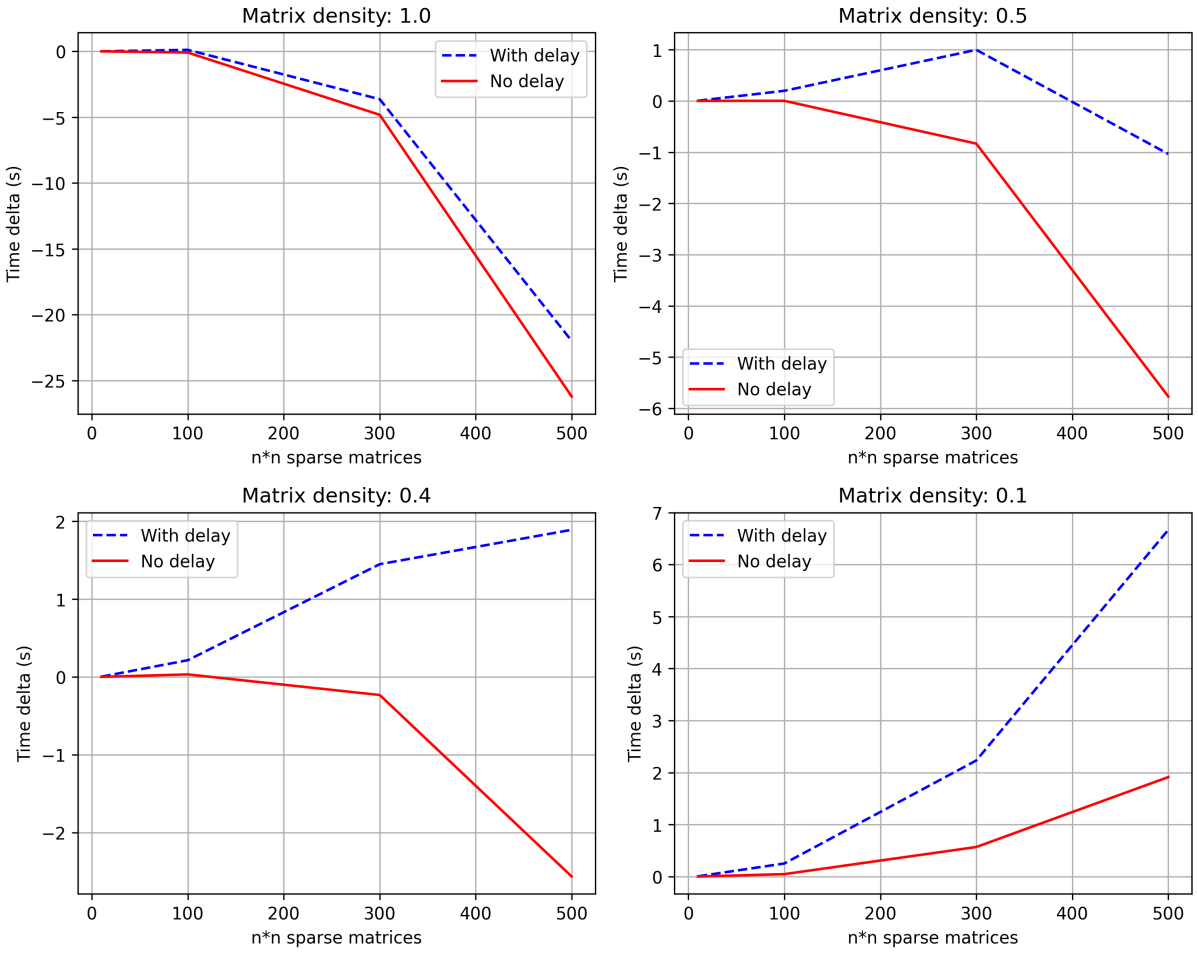

Figure 1: Difference in runtime between Python (NumPy) and SQL for matrix multiplication. $\Delta t > 0$ means SQL was faster than Python. The dashed line includes a simulated 10 Mbps connection penalty applied to the Python workflow.

3.1 Dense matrices

- At high densities (large nonzero proportion d), we see that Python outperforms SQL by a wide margin and this gap grows with matrix size. This is due to the highly optimised nature of the NumPy algorithms.
- The simulated network connection makes little difference since the SQL approach is so slow.

3.2 Sparse matrices

- At low densities (small nonzero proportion d), SQL becomes much faster as it can filter out most operations (since they are multiplication by 0). NumPy, on the other hand, is still multiplying everything out so is slower.
- The delay now makes much more of a difference since the SQL approach is far more competitive and in fact wins as matrices become sparser (smaller d). This being said the absolute delay is the same, it is only relatively that this is changing.

4 Future Work

Concrete next steps that would make this study stronger:

True sparsity at present, we store all values with no implied 0s, this is inefficient and increases the effect of the delay for sparse matrices.

Larger n test larger n until we reach the limits of memory so that we must use disk storage, I hypothesize that this would further favour the SQL approach.

Find the boundary We do not know exactly where the boundary is between SQL or Python being faster as we vary sparsity.