# Benchmarking SQL and NumPy for Matrix Multiplication

Tristan Hodgson

December 21, 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

Both matrices and relational databases are ubiquities in many applications such as data science and graph theory. In these cases we often need to multiply matrices stored in such databases. In this paper we attempt to describe when it is better to do multiplication of matrices in SQL compared to a more traditional approach using a programming language like Python.

We anticipate that the more traditional approach may be advantageous for some matrices due to the ability to leverage highly optimised libraries such as NumPy. However, some matrices may benefit from the SQL approach as we do not have to export all of our data from the database into Python and then back again. We also investigate how a slow internet connection (between the server and client) may affect the relative speed of each approach.

Our findings apply to the case where

- Our matrices are stored in an SQL database to begin with
- We want the product of two matrices to be stored in the SQL database after it has been calculated

## 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 *non-zero proportion*), each entry is kept with probability $d$ and set to 0 otherwise. To increase speed we only randomly generate the largest size matrix and then truncate it to the top left corner as we need smaller matrices. Using this approach we need only generate new matrices for each repeat of the experiment and for each density.

**SQL Multiplication** Matrix multiplication can 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. This translation allows us to take advantage of the parallelisation of SQL with little to no work on our part.

```
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;
```

**Python Multiplication** For multiplying in Python we use NumPy's built in function. For each test we 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 field work in a low connectivity environment. This speed also has the advantage of making the result more pronounced meaning we can more easily see the effect of slow internet speed.

The Python approach is the only one that needs a delay as it downloads $A$ and $B$ and then uploads $(AB)$, while no substantial data transfers occur under SQL. 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 [1], the simulated transfer time is

$$t_{\text{delay}} = \beta(nm + mp + np) \qquad \beta := \frac{64}{S \cdot 10^6}$$

where $S$ is the internet 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/overheads) in the delay model. Ignoring these effects makes Python more competitive compared to SQL.

**Benchmarking** We benchmark random square matrices with $n \in \{100, 300, 500, 700\}$ and $d \in \{0.1, 0.3, 0.4, 0.5, 1\}$ . We plot $\Delta t = t_{\text{Py}} - t_{\text{SQL}}$ averaged across three runs. As such $\Delta t > 0$ means SQL is faster, and $\Delta t < 0$ means Python is faster. We use square matrices for simplicity only.

## 3 Modelling

We expect matrix multiplication to take $O(n^3)$ time, where n is the size of the matrix. This is the approach that NumPy will take. We also add $O(n^2)$ time due to the simulated transfer cost. We therefore model the time taken as $\alpha n^3 + \beta n^2$ where $\alpha$ is an unknown constant and, since we are working with square matrices we abuse notation to say that, $\beta = \frac{192}{S \cdot 10^6}$.

In contrast, SQL filters out all zeros before doing our multiplication so we do $O(n \times dn \times dn)$ work where $d$ is the density of non-zero entries. This is because, $P(a_{ik} \neq 0) = P(b_{kj} \neq 0) = d \implies P(a_{ik} \neq 0 \cap b_{kj} \neq 0) = d^2$ by independence. So we can model the time as $\gamma d^2 n^3$.

Hence to find when SQL is faster we need to find when $\alpha n^3 + \beta n^2 > \gamma d^2 n^3$ which is true when $d < d_{\text{crit}} = \sqrt{\dfrac{\alpha + \frac{\beta}{n}}{\gamma}}$ (since all quantities are positive by assumption). Now all that remains to be able to find this boundary concretely is the estimate the values of $\alpha, \gamma$.

### 3.1 Estimating constants

We estimate the constants using a linear regression model. See the proofs in the appendix for the formulae below.

$$\hat{\alpha} = \frac{\sum_{i=1}^{N} n^3 t_{\text{Py}}}{\sum_{i=1}^{N} n^6}$$

$$\text{SE}(\hat{\alpha}) = \sqrt{\frac{\sigma_{\text{Py}}^2}{\sum_{i=1}^{N} n^6}}$$

$$\sigma_{\text{Py}}^2 = \frac{1}{N-1} \sum (t_{\text{py}} - \hat{\alpha} n^3)^2$$

$$\hat{\gamma} = \frac{\sum_{i=1}^{N} d^2 n^3 t_{\text{SQL}}}{\sum_{i=1}^{N} d^4 n^6}$$

$$\text{SE}(\hat{\gamma}) = \sqrt{\frac{\sigma_{\text{sql}}^2}{\sum_{i=1}^{N} d^4 n^6}}$$

$$\sigma_{\text{sql}}^2 = \frac{1}{N-1} \sum (t_{\text{SQL}} - \hat{\gamma} d^2 n^3)^2$$

| Estimator | Estimate | SE | % error | 95% confidence interval |
|---|---|---|---|---|
| $\hat{\alpha}$ | $7.969 \times 10^{-8}$ | $2.444 \times 10^{-9}$ | 3.066 | $(7.490 \times 10^{-8}, 8.448 \times 10^{-8})$ |
| $\hat{\gamma}$ | $6.178 \times 10^{-7}$ | $1.614 \times 10^{-8}$ | 2.613 | $(5.861 \times 10^{-7}, 6.494 \times 10^{-7})$ |

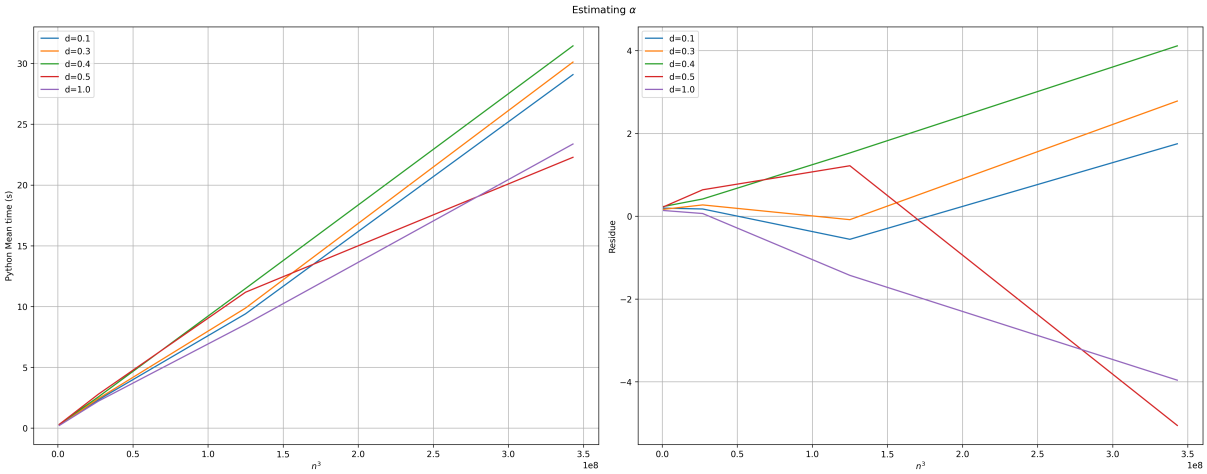


Figure 1: Plots of the time taken for Python and residuals as n varies

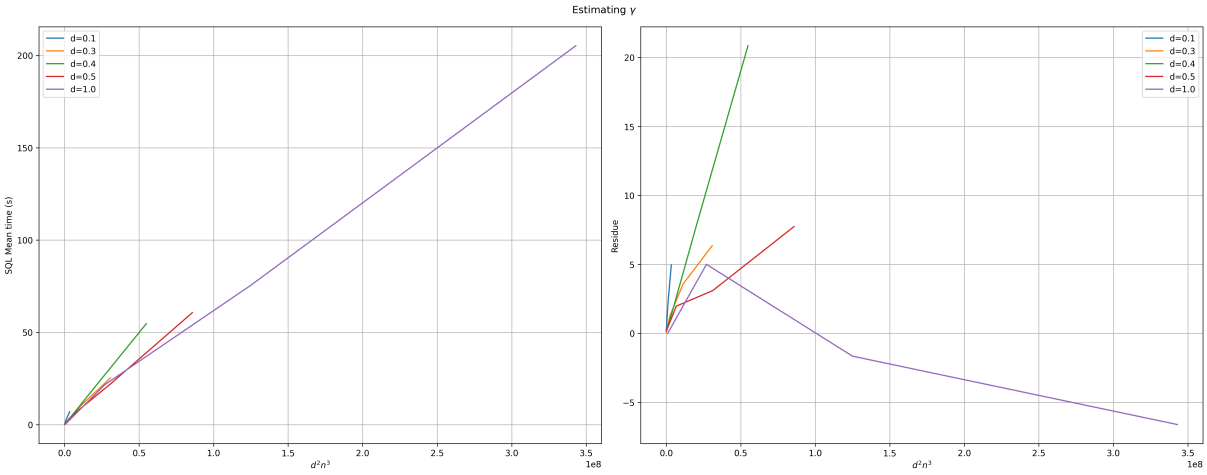


Figure 2: Plots of the time taken for SQL and residuals as n varies

Looking at the residuals and standard error, we see that our model is at least reasonable and does somewhat accurately model the data. That being said we can also see that residuals are not constant with the x-axis variable on both the $\alpha$ and $\gamma$ plots. This casts significant doubt on the foundations of our model. That being said, we will still accept it since it does have strong prediction power on the data we consider.

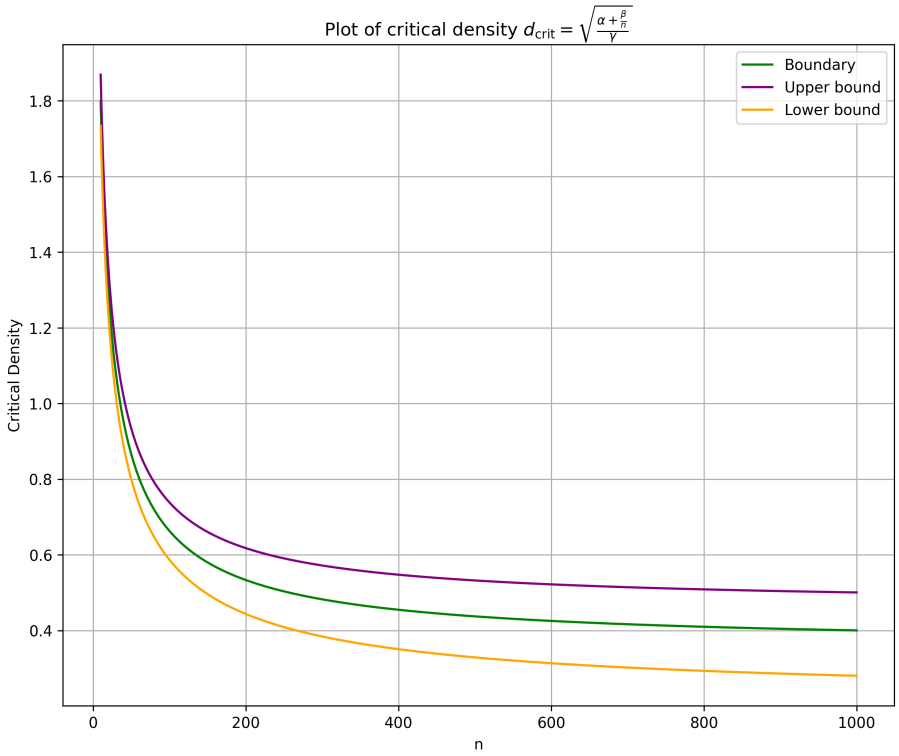


Figure 3: Plot of how $d_{\text{crit}}$ varies as $n$ varies

When the density is less than $d_{\text{crit}}$ SQL will be faster, this corresponds to the region below the green boundary line in figure 3. Note that since we interpret $d_{\text{crit}}$ as a density, it only makes sense if $d_{\text{crit}} \in [0, 1]$, so our model predicts that for $d < 36$ there is no $n$ for which SQL can win.

## 4 Results

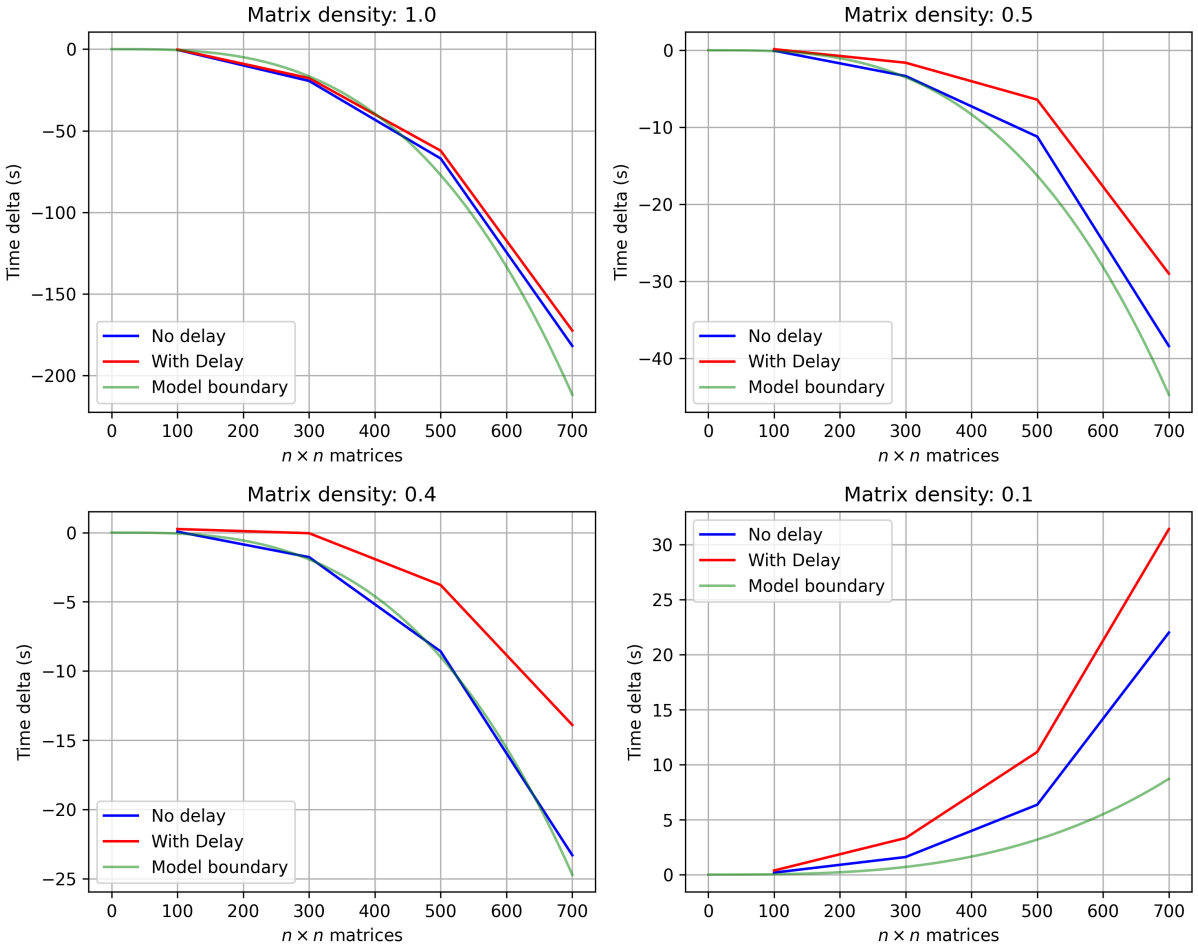


Figure 4: Difference in runtime between Python (NumPy) and SQL for matrix multiplication. $\Delta t > 0$ means SQL was faster than Python. The red line includes a simulated 10 Mbps connection penalty applied to the Python workflow. The green line shows what our model predicts.

### 4.1 Dense matrices

- At high densities, 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.
- We get strong agreement with our model (by eye) with it strongly tracking the red line.

### 4.2 Sparse matrices

- At low densities, 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.
- We get much weaker agreement with our model (by eye) with it diverging from the red line, this suggests our model is weaker in this range.

## 5 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.

**Structured matrices** Many applications of this result may use matrices that are not randomly dense i.e. the Bernoulli random mask that we applied may not be appropriate. For example, a page rank adjacency matrix would display large amounts of clustering. This may affect our results.

**Use GPU** Compare other libraries including those that enable GPU acceleration such as PyTorch

**Increase the number of runs** Increasing the number of runs would decrease the run-to-run variance of the estimates for $\alpha$ and $\gamma$.

**Non-square** Using non-square matrices (e.g. vectors for a dot product) could very meaningfully change our results but we have not considered this here. Square matrices are perhaps the most common in applications (e.g. Markov chains, adjacency matrices) hence we consider only these.

**Explore better models** We saw that our model's predictions did not track the observed data, particularly at lower densities. Other models, for example including lower order terms, could produce better results. It is interesting (though perhaps insignificant) that our model seems to lower bound the time; perhaps this suggests that it comes from us ignoring overhead in the transfer time.

## 6 Appendix: Proof of regression formulae

**Claim:** $\hat{\alpha} = \dfrac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$

$$S(\alpha) := \sum_{i=1}^{N} (y_i - \alpha x_i)^2$$

$$\hat{\alpha} = \operatorname{argmin}\ S(\alpha)$$

$$\frac{\partial S(\alpha)}{\partial \alpha} = -2 \sum_{i=1}^{N} x_i (y_i - \alpha x_i)$$

$$0 = \sum_{i=1}^{N} x_i (y_i - \hat{\alpha} x_i)$$

$$\hat{\alpha} = \frac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$$

**Claim:** $\operatorname{Var}(\hat{\alpha}) = \dfrac{\sigma^2}{\sum_{i=1}^{N} x_i^2}$

$$\begin{aligned}
\operatorname{Var}(\hat{\alpha}) &= \operatorname{Var}\left( \frac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2} \right) \\
&= \operatorname{Var}\left( \frac{\sum_{i=1}^{N} x_i (\alpha x_i + \epsilon_i)}{\sum_{i=1}^{N} x_i^2} \right) \\
&= \operatorname{Var}\left( \alpha + \frac{\sum_{i=1}^{N} x_i \epsilon_i}{\sum_{i=1}^{N} x_i^2} \right) \\
&= \frac{1}{(\sum_{i=1}^{N} x_i^2)^2} \operatorname{Var}\left( \sum_{i=1}^{N} x_i \epsilon_i \right) \\
&= \frac{1}{(\sum_{i=1}^{N} x_i^2)^2} \left( \sum_{i=1}^{N} x_i^2 \operatorname{Var}(\epsilon_i) \right) \\
&= \frac{\sigma^2}{\sum_{i=1}^{N} x_i^2} \qquad \because \epsilon_i \sim N(0, \sigma^2) \text{ by assumption}
\end{aligned}$$

Both claims are analogous for $\gamma$.

## References


[1] Postgresql docs. https://www.postgresql.org/docs/current/datatype-numeric.html. Accessed 2025-12-19.