Database is All You Need: Serving LLMs with Relational Queries

Wenbo Sun
w.sun-2@tudelft.nl
Delft University of Technology
the Netherlands

Ziyu Li
zli17nl@gmail.com
Delft University of Technology
the Netherlands

Vaishnav Srinidhi
v.srinidhi@student.tudelft.nl
Delft University of Technology
the Netherlands

Rihan Hai
r.hai@tudelft.nl
Delft University of Technology
the Netherlands

ABSTRACT

Large language models (LLMs) have become central to many applications, but their deployment often requires high-performance hardware, specialized libraries, and complex engineering, limiting accessibility for smaller organizations. Meanwhile, relational database systems (RDBMS) are widely used for portability, efficiency, and native support for managing large-scale data operations.

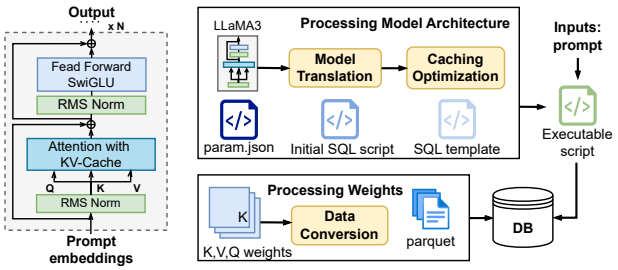
This paper presents *TranSQL*[1], a toolkit that enables transformer-based LLM inference within RDBMS. By translating neural operations into SQL queries and representing model weights as relational tables, *TranSQL* leverages database features like dynamic disk-to-memory data management and caching to reduce hardware and engineering demands for serving LLMs. Using the LLaMA3 8B model, we demonstrate *TranSQL*'s ability to implement attention layers, KV-cache, and end-to-end text generation through SQL queries. *TranSQL* offers a cost-effective, portable, and scalable approach to making advanced AI technologies more accessible.

1 INTRODUCTION

Large language models (LLMs) achieve exceptional performance but often demand significant resources, including hardware accelerators, specialized linear algebra libraries, and advanced engineering techniques, e.g., KV-caching [12], tensor parallelism [15], and memory-disk offloading [3], to manage their computational and memory requirements. These demands present challenges for scaling and accessibility, particularly for organizations lacking the necessary infrastructure or expertise.

At its core, LLM computation relies on a series of linear algebra operations, primarily matrix multiplications (MM) and vector transformations, which align naturally with relational database capabilities. For instance, MM can be decomposed into chunk-based operations, implemented as relational queries [7, 13], mirroring tensor parallelism by distributing computations across smaller chunks.

This compatibility extends to other optimization techniques. KV-caching, a critical feature in LLMs, aligns naturally with database functionalities such as in-memory tables (`pg-mem`[2] in PostgreSQL) and dictionaries[3] in ClickHouse. Similarly, dynamic disk-to-memory data management—a technique for handling large-scale LLMs when model weights exceed available memory—is an inherent capability of disk-based database systems. Additionally, weights and activations in neural networks can be effectively represented as tabular data [11], further highlighting the natural alignment between LLM computations and relational database management systems (RDBMS).



(a) Transformer unit (b) TranSQL architecture with three major components

Figure 1: (a) The architecture of a single transformer unit, illustrating the key components such as attention with KV-cache, feed-forward layers. (b) Overview of the *TranSQL* toolkit, which consists of three major components: Data conversion, model translation and caching management.

The alignments suggest that RDBMS are not only capable of managing LLM computations but also offer opportunities to integrate optimization techniques like caching and parallelism directly into the database ecosystem. Rather than competing with GPUs or deep learning frameworks, this work introduces an alternative approach to leveraging databases' versatility and portability for LLM deployment.

This paper presents *TranSQL*, a novel toolkit for serving transformer-based models entirely within RDBMS. By leveraging the alignment between LLM operations and database functionalities, *TranSQL* provides a portable and economic solution that redefines how LLMs can be deployed.

We summarize three major contributions, corresponding to the three components of *TranSQL*, as illustrated in Figure 1(b):

- **Data conversion**: Model weights are sliced into equal-size vectors and stored as relational tables, enabling seamless integration with database features like vectorized execution and SIMD instructions in CPUs.
- **Model translation**: LLM inference is mapped to relational queries, allowing core neural operations such as matrix multiplications and attention mechanisms (Figure 1(a)) to

[1]We illustrate the demo scenarios in a short video. https://drive.google.com/file/d/1PWrvVf8De3t5HgwZ-x3x6ONCtzgG36Su/view?usp=sharing
[2]https://github.com/oguimbal/pg-mem
[3]https://clickhouse.com/docs/en/sql-reference/dictionaries

be executed directly within the database. This integration avoids the need to develop custom optimizations, instead leveraging the well-optimized SQL ecosystem.

- **KV-Cache management**: By adapting native database caching mechanisms, we implement KV-cache functionality for efficient reuse of intermediate results during token generation, significantly improving inference performance.

These contributions address critical challenges in LLM deployment by utilizing built-in database features to achieve functionalities like KV-caching, tensor parallelism, and memory-disk offloading without the need to re-design these optimizations for LLMs. Operating entirely within the database ecosystem, *TranSQL* leverages the inherent scalability, portability, and efficiency of RDBMS, offering a flexible framework adaptable to other transformer-based models and even diffusion models [17] with minimal modifications.

We demonstrate *TranSQL* using the LLaMA3 8B model [4], converting its weights into relational tables and employing ClickHouse[4], a high-performance RDBMS with vector computation support. The model's inference process is fully translated into SQL queries interacting with these tables. This end-to-end demonstration illustrates the feasibility and potential of running large-scale LLMs entirely within an RDBMS environment, opening new avenues for scalable and resource-efficient AI deployment.

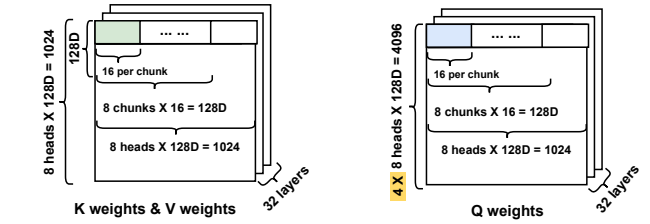
2 RELATED WORK

In-Database Machine Learning (ML). A common approach to integrating ML with databases relies on user-defined functions (UDFs), as seen in frameworks like MADlib [5] and PostgresML[5]. While these enable model training and serving through SQL queries, UDFs act as black boxes, requiring separate runtimes and lacking transparency for database optimizers. Similarly, commercial databases like SAP HANA [2] and IBM DB2 [1] provide limited native model-serving capabilities, but these rely on external training and proprietary vendor support, making them costly and less accessible. In contrast, our approach rewrites ML operations into native SQL, eliminating the need for external runtimes or libraries. This ensures greater portability, extensibility, and broader accessibility. Another line of research rewrites ML algorithms as relational queries, as seen in frameworks like F [13] and AC/DC [7]. While these focus on factorized learning for traditional, primarily linear models, they require hand-crafted rewrites for each model, limiting extensibility. In contrast, our approach targets core operators in transformer models, such as attention and feedforward layers, enabling support for non-linear models and offering greater scalability for modern deep learning architectures.

Deep Learning in RDBMS. Recent efforts in the database community have explored representing deep learning computations using relational algebra. Systems like DuckBrain [14] and DL2SQL [9] convert neural operators into relational queries but face scalability challenges due to quadratic growth in tuples as models scale. Some research, like Dimitrije et.al. [6], SmartLite [10], and ModelJoin [8] use matrix types and operations in RDBMSs but require significant engineering and optimized algorithms for efficiency.

Our approach balances scalability and simplicity by representing matrices as vectors, leveraging existing RDBMS support for arrays and lambda functions. This avoids extensive engineering while allowing integration with linear algebra libraries or hardware accelerators for enhanced performance.

[4]https://clickhouse.com/
[5]https://github.com/postgresml/postgresml



layer	head	row	col	key_chunk	value_chunk
0	0	0	0	<0.6,0.4...>	<0.2,0.3...>
0	0	0	1	<0.7,0.3...>	<0.6,0.5...>

layer	head	row	col	query_chunk
0	0	0	0	<0.5,0.4...>
0	0	0	1	<0.7,0.1...>

Figure 2: The Q, K, V weights are sliced into chunks of 16-element vectors and stored in relational tables with indices.

3 TRANSQL: SERVING LLMS USING RELATIONAL QUERIES

In this section, we introduce the major components of TranSQL, our framework for serving large language models (LLMs) using relational queries. This section is organized as follows: we begin by explaining how *TranSQL* converts tensors into relational tables. Next, we describe how matrix multiplications are performed using relational queries on this data and extend the discussion to primary neural operations in transformer-based LLMs. Finally, we outline the implementation of a key-value (KV) caching mechanism in SQL to accelerate token generation during inference.

3.1 Relational Representation for Transformer Models

3.1.1 Representing Model Weights. To represent model weights as relational table, TranSQL slices matrices into rows of tiled vectors and storing them in relational tables, as shown in Figure 2. Each row of a matrix is partitioned into equal-size vectors (or chunks). The relational representation of a $m \times n$ matrix $M \in \mathbb{R}^{m \times n}$ is a set of tuples T.

For transformer models, the weight matrices have additional structural information beyond simple matrices. Taking the LLaMA3 8B model as an example, weights in the attention mechanism are organized hierarchically by layers and heads. Additionally, LLaMA3 uses grouped attention, where each key ($\mathbf{k}$) and value ($\mathbf{v}$) vector corresponds to **four** query ($\mathbf{q}$) vectors. To represent this, we introduce two additional indices: the layer index l, and head index h. Thus, the weight representation for attention heads is defined as follows:

$$T = \{(l, h, r, c, \mathbf{v}) \mid l \in [0, L), h \in [0, H), \\ r \in [0, m), c \in [0, \lceil n/t \rceil), \mathbf{v} \in \mathbb{R}^t\}, \tag{1}$$

where: r is the row index (integer), c is the column tile index (integer), $\mathbf{v}$ is a vector chunk of size t (e.g., $t = 16$), derived by slicing the matrix row at $c \cdot t$ to $(c + 1) \cdot t$, and L and H denote the number of layers and attention heads, respectively.

Determining Chunk Size. To optimize vector operations on CPUs, we utilize SIMD instructions like AVX512, which enable parallel processing of multiple floating-point numbers in a single operation (e.g., 16 single-precision numbers with AVX512). Accordingly, we set the chunk size t to align with the SIMD vector width, maximizing hardware efficiency. Our data converter

Table 1: Primary neural operations and corresponding pseudo SQL queries.

LLaMA3 layers	Notation	Neural operations	SQL query
Attention	$softmax(\frac{Q \cdot K^T}{\sqrt{d}})V$	Matmul Matmul Matmul	`SELECT token, head, row, SUM(DOT(query_chunk, embedding)) AS q FROM query[key,value]_weights as A JOIN embedding as B ON A.col = B.col GROUP BY row`
		Matmul	`SELECT Q.token, K.token, Q.head, EXP(SUM(q * k) / sqrt(head_dim)) as qk FROM Q JOIN K on Q.row = K.row and Q.head//4 = K.head GROUP BY Q.token, K.token, Q.head`
		Softmax	`WITH summation AS ( SELECT Q.head, Q.token, SUM(qk) as s FROM QK GROUP BY Q.token, Q.head ) SELECT Q.head, Q.token, K.token, qk/s FROM QK JOIN sumamtion ON Q.head, Q.token`
Rotary positional encoding	$\mathbf{x} = \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^{d/2}$	Split as complex	`SELECT token, head, COLLECT_COMPLEX(groupArray(row), groupArray(r), 0) AS` x_1, `COLLECT_COMPLEX(groupArray(row), groupArray(r), 1) AS` x_2 `FROM` x `GROUP BY token,head`
	$\begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix}$	Rotation	`SELECT token, head, HADAMARD(`x_1`, `θ_1`) - HADAMARD(`x_2`, `θ_2`) as real, HADAMARD(`x_1`, `θ_2`) + HADAMARD(`x_2`, `θ_1`) AS img FROM complex_vectors as A LEFT JOIN` $rotation_\theta$ `AS B ON A.token=B.token`
	Concat$(\mathbf{x}_1^{(p)}, \mathbf{x}_2^{(p)})$	Merge to vector	`SELECT layer, token,head, VIEW_AS_REAL(real, img) as chunk FROM positional_encoding`
RMS Norm	$\frac{1}{\sqrt{(\sum \mathbf{x}^2/\text{dim})}}$	Squared mean & rsqrt	`Sellect token, 1/SQRT(SUM(arraySum(`x`->`x^2`,embedding)) /` dim`) + `ϵ`) as rep_sqmean FROM Embedding GROUP BY token`
	rep_sqmean*x*norm_weight	Weighted mean	`SELECT token, col, rep_sqmean * HADAMARD(`x`, weight) AS new_embedding FROM Embedding LEFT JOIN rsqrt ON rsqrt.token= Embedding.token LEFT JOIN norm_weight ON Embedding.col = norm_weight.col`

automatically detects the machine's supported SIMD instruction set and partitions matrices into chunks accordingly. For example, with AVX512, matrices are sliced into chunks of 16 floating-point numbers.

3.1.2 Representing Computations. The core of translating LLM inference into SQL lies in performing matrix multiplication using relational algebra. With tensors represented in relational tables as shown in Eq. 3.2.1, matrix multiplication for specific l and h can be expressed as:

```
SELECT l, h, t1.r, SUM(DOT(t1.v, t2.v))
FROM t1 JOIN t2 ON t1.c = t2.c GROUP BY t1.r;
```

Building on this foundation, other neural operations, such as attention layers and feedforward layers, are implemented. Softmax and normalization require additional aggregation steps, while rotary positional encoding [16] involves complex number arithmetic. Table 1 provides an overview of primary neural operations and their corresponding SQL queries. The queries in the table are pseudo-code representations, and to support all required operations, we introduced custom vector functions, including merging complex numbers into a single vector (`VIEW_AS_REAL`), splitting vectors into complex numbers (`COLLECT_COMPLEX`), and Hadamard product (`HADAMARD`),

3.2 Key-Value Cache Implementation

Key-value (KV) caching [12] is essential for efficient LLM inference. Attention layers calculate attention scores using all past tokens as keys (**K**) and values (**V**). Without caching, these computations are repeated for every token, leading to quadratic complexity $O(n^2)$ as sequence length (n) increases. KV-caching reduces this to $O(n)$ by storing precomputed keys and values, significantly accelerating generation.

Many databases support in-memory tables for frequent access. PostgreSQL offers pg-mem, MySQL includes a memory engine, and ClickHouse provides dictionaries with composite key indexing. In our approach, we leverage ClickHouse's dictionary to implement KV-caching for efficient LLM inference.

3.2.1 KV-Cache in SQL. In *TranSQL*, we implement the KV-cache mechanism using the relational database's native capabilities, such as table indexing and efficient incremental updates. The implementation consists of the following components:

Cache Storage. The cached keys and values are stored in a dedicated table:

$$Cache = \{(l, i, h, \mathbf{k}, \mathbf{v}) \mid l \in [0, L], i \in [0, I), h \in [0, H), \mathbf{k}, \mathbf{v} \in \mathbb{R}^t\},$$

where i denotes the token index, h the head index, $\mathbf{k}$ the cached key vector, and $\mathbf{v}$ the cached value vector. Since the $\mathbf{k}$ and $\mathbf{v}$ values are used layer by layer, *TranSQL* preloads these values to a dictionary for each layer at the start of the corresponding layer's computation. This preloading process can be performed asynchronously, improving efficiency by overlapping with other query executions.

Cache Updates. During the generation of a new token, the key ($\mathbf{k}_{\text{new}}$) and value ($\mathbf{v}_{\text{new}}$) vectors for the new token are computed. These vectors are appended to the **Cache** table via an `INSERT`.

Efficient Querying. Key and value vectors are preloaded into a dictionary, enabling efficient retrieval via `SELECT` queries indexed by head and token indices. These cached vectors are combined with the current token's key and value vectors to compute attention scores, extending matrix multiplication queries with additional joins. By incorporating KV-cache into the relational queries, *TranSQL* eliminates the need for specialized caching protocols, instead leveraging the database's built-in indexing and optimization mechanisms for scalable and efficient caching.

4 DEMONSTRATION SCENARIOS

The demonstration consists of three components: i) Interactive Frontend: A user interface for interacting with the system and viewing inference results. ii) TranSQL: The core framework that converts model data and inference into SQL queries. iii) Relational Database: The backend system executing the translated SQL queries.

We showcase the demonstration with the LLaMa3 8B model and ClickHouse, running on an AWS c7i.2xlarge instance. The demonstration includes two scenarios: i) Display the SQL queries for serving the model. ii) Input a prompt to generate tokens.

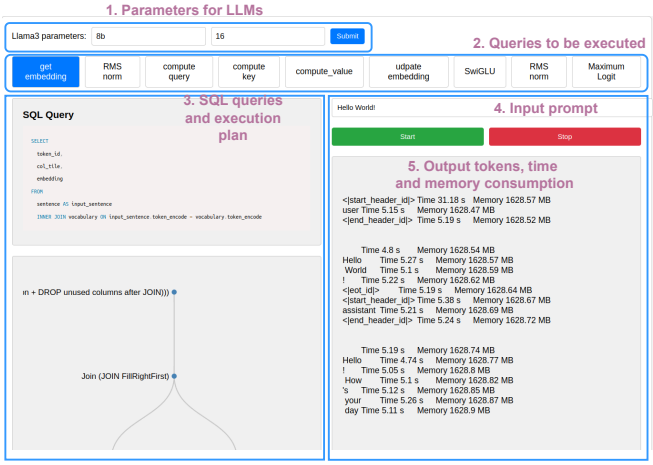


Figure 3: Screenshot of the demonstration interface. Participants specify the model scale and tile size in Block 1. Block 3 shows SQL queries with their logical execution plans, while the Block 5 displays tokens generated line by line based on the input prompt.

4.1 Relational Data and Query Translation

The demonstration begins with loading the LLaMA3 model into the database, guided by the scale and tile size specified by participants in Block 1. *TranSQL* splits tensor weights into equal-sized vectors and generates indices, which are stored in Parquet files as an efficient intermediate representation. These Parquet files are then imported into the ClickHouse database as relational tables, organizing the model's weights into a tabular format suitable for SQL-based operations. Simultaneously, the operations required for LLM inference are translated into a sequence of SQL queries, as shown in Block 2, designed to operate on the imported data.

After the conversion process is complete, participants interact with the system through Block 3 of the visual interface (Figure 3). The interface supports two key activities: i) viewing the execution plans for translated SQL queries to gain insights into how the database processes LLM inference operations, and ii) exploring interactively by selecting specific layers of the LLaMA3 8B model to examine the associated SQL queries and weights stored in the relational tables.

4.2 Input Prompt and Get Response

In the second part of the demonstration, participants observe how the system performs inference using the weights and SQL queries generated in the previous step.

Participants can input custom prompts in Block 4 and initiate the inference process by clicking the start button. The system dynamically generates subsequent tokens. During this process, participants can view the time consumption and peak memory usage of the database for each token in Block 5, as shown in Figure 3. This feature highlights the memory footprint of SQL-based LLM inference, demonstrating how relational database capabilities efficiently manage resources.

5 CONCLUSION AND DISCUSSION

In this paper, we introduced *TranSQL*, a toolkit for performing transformer-based LLM inference within relational database systems (RDBMS). By translating tensor operations into SQL queries and storing model weights as relational tuples, *TranSQL* leverages database capabilities like dynamic disk-to-memory management and native caching. This approach offers an alternative for deploying large-scale language models in environments with limited access to specialized infrastructure.

Using the LLaMA3 8B model, we demonstrated *TranSQL*'s ability to implement core neural operations, including matrix multiplication, attention mechanisms, and KV-cache, enabling end-to-end text generation within an RDBMS. The toolkit's adaptability allows it to support other transformer-based architectures and diffusion models with minimal modifications. Future work will focus on scaling *TranSQL* for larger models, optimizing database-specific performance, and extending compatibility across diverse RDBMS platforms.

REFERENCES

[1] 2023. In-database Machine Learning. https://www.ibm.com/docs/en/db2/12.1?topic=content-in-database-machine-learning.

[2] 2024. SAP HANA PAL documentation. https://help.sap.com/docs/SAP_HANA_PLATFORM/2cfbc5cf2bc14f028cfbe2a2bba60a50/c9eeed704f3f4ec39441434db8a874ad.html?version=2.0.07.

[3] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, and et.al. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. In *ACL'24*. Bangkok, Thailand, 12562–12584.

[4] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). arXiv:2407.21783

[5] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1700–1711.

[6] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, et al. 2020. Declarative Recursive Computation on an RDBMS: or, Why You Should Use a Database For Distributed Machine Learning. *SIGMOD Rec.* 49, 1 (2020), 43–50.

[7] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *DEEM'18*. Article 8, 10 pages.

[8] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *EDBT'23*. 311–323.

[9] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1794–1807.

[10] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-based Serving System for DNN Inference in Resource-constrained Environments. *Proc. VLDB Endow.* 17, 3 (2023), 278–291.

[11] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, et al. 2017. Scalable Linear Algebra on a Relational Database System. In *ICDE'17*. 523–534.

[12] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *MLSys'23*.

[13] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD'16*. 3–18.

[14] Maximilian E. Schüle, Thomas Neumann, and Alfons Kemper. 2024. The Duck's Brain. *Datenbank-Spektrum* 24, 3 (2024), 209–221.

[15] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. abs/1909.08053 (2019). arXiv:1909.08053

[16] Jianlin Su et al. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[17] Ling Yang et al. 2023. Diffusion models: A comprehensive survey of methods and applications. *Comput. Surveys* 56, 4 (2023), 1–39.