TranSQL$^+$: Serving Large Language Models with SQL on Low-Resource Hardware

WENBO SUN, Delft University of Technology, Netherlands
QIMING GUO, Texas A&M University - Corpus Christi, USA
WENLU WANG, Texas A&M University - Corpus Christi, USA
RIHAN HAI, Delft University of Technology, Netherlands

Deploying Large Language Models (LLMs) on resource-constrained devices remains challenging due to limited memory, lack of GPUs, and the complexity of existing runtimes. In this paper, we introduce **TranSQL$^+$** , a template-based code generator that translates LLM computation graphs into pure SQL queries for execution in relational databases. Without relying on external libraries, TranSQL$^+$ leverages mature database features—such as vectorized execution and out-of-core processing—for efficient inference. We further propose a row-to-column (ROW2COL) optimization that improves join efficiency in matrix operations. Evaluated on Llama3-8B and DeepSeekMoE models, TranSQL$^+$ achieves up to 20× lower prefill latency and 4× higher decoding speed compared to DeepSpeed Inference and `Llama.cpp` in low-memory and CPU-only configurations. Our results highlight relational databases as a practical environment for LLMs on low-resource hardware.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Large Language Models, Relational Database, Query Processing

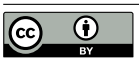
ACM Reference Format:
Wenbo Sun, Qiming Guo, Wenlu Wang, and Rihan Hai. 2025. TranSQL$^+$: Serving Large Language Models with SQL on Low-Resource Hardware . *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 371 (December 2025), 27 pages. https://doi.org/10.1145/3769836

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities, yet they are predominantly deployed on powerful cloud servers with high-end GPUs and abundant memory resources. The cloud-centric paradigm raises privacy concerns and complicates integration with local tools. When a task requires local tooling, one must either expose the local environment [13, 49] so a remote service can produce tool-compatible outputs or accept reduced functionality—undermining both privacy and customizability. These drawbacks motivate performing LLM inference directly on resource-constrained personal or edge devices. With appropriate fine-tuning or distillation, edge deployments can provide sufficient performance for many specialized applications. For instance, multiple studies in healthcare [26, 29] have demonstrated that smaller, domain-specific models can achieve superior accuracy compared to large general-purpose models, enabling specialized deployments such as on-device diagnostic tools or personal health assistants.

Nevertheless, edge devices—ranging from personal computers to billions of smartphones and IoT endpoints—typically lack GPUs, have limited memory, and operate within fragmented software and hardware ecosystems. Crucially, even smaller domain-specific models can exceed the memory of

Authors' Contact Information: Wenbo Sun, Delft University of Technology, Netherlands, w.sun-2@tudelft.nl; Qiming Guo, Texas A&M University - Corpus Christi, USA, qguo2@islander.tamucc.edu; Wenlu Wang, Texas A&M University - Corpus Christi, USA, wenlu.wang@tamucc.edu; Rihan Hai, Delft University of Technology, Netherlands, r.hai@tudelft.nl.



ACM 2836-6573/2025/12-ART371
https://doi.org/10.1145/3769836

typical edge devices. For example, Llama3 8B [15], which is often praised for its performance–size tradeoff, requires 16.1GB of RAM when loaded in memory, exceeding the 8–16GB available on most laptops and smartphones. This constraint complicates efficient single-batch inference under memory limitations, raising a significant challenge for practical edge-based LLM deployments.

A typical LLM deployment on edge devices starts with model compression, followed by compilation to map neural operators to hardware-specific tensor backends. Most setups load the entire model into memory for fast inference, but this is often infeasible on devices with limited RAM. To address this, various techniques have been proposed—each with trade-offs. Compression methods (e.g., distillation, quantization, pruning) reduce memory usage but can degrade accuracy and require tuning [9, 46]. Offloading weights between GPU and CPU memory [27, 37, 43] is another strategy, though mainly optimized for GPU systems. Few frameworks support weight offloading directly to NVMe storage. The frameworks, such as `Llama.cpp`, rely on memory-mapped I/O to support swapping active weights from disk to memory dynamically [6, 9, 14, 37]. However, NVMe offloading in these systems often introduces significant I/O bottlenecks and inefficient disk access patterns, severely degrading inference performance when memory is constrained.

In addition, the heterogeneity of edge environments complicates deployment. Personal computers are predominantly x86_64, whereas embedded and edge devices span ARM v8/v9 [5], RISC-V [7], and other ISAs. Although deep learning compilers (e.g., ONNX [1], TVM [11], MLIR [28]) can compile models to common intermediate representations, supporting new operators and heterogeneous hardware still demands re-engineering and related skills: developers must implement ISA-specific kernels and instruction support, then ensure every target device matches up-to-date runtime. Consequently, porting and optimizing inference frameworks across such diverse hardware remains complex and expensive.

Our proposal. To address these challenges, we propose a fundamentally different approach: leveraging the ubiquity and maturity of **relational databases** to serve as inference engines. Databases such as SQLite are embedded in billions of devices worldwide [2], while modern database engines offer robust out-of-core execution, efficient cache management, and vectorized execution [20, 36]. Databases align well with the demands of interactive workloads such as LLM inference under tight memory budgets, efficiently managing model parameters that exceed RAM and accelerating linear-algebra kernels via batch-oriented, vectorized execution. The same buffer-management and I/O-scheduling advantages benefit streaming workloads—for example, object detection with convolutional neural networks (CNNs)—making databases a strong fit for resource-constrained deployments.

Moreover, SQL's inherent portability enables deployment across diverse platforms without architecture-specific tweaks or external ML runtimes. Prior work has shown that relational databases can execute neural computations when cast in relational form [22, 31, 39], confirming their viability as inference engines. In practice, this means developers with basic database skills can serve models using familiar SQL operations, rather than learning or maintaining specialized compiler stacks.

Building on these observations, we introduce TranSQL$^+$, a template-based code generator that converts LLM computation graphs into pure SQL queries executable entirely within a relational database. The forward pass of the model—including matrix multiplications, attention, and non-linear activations—is expressed using relational operations such as joins and aggregations. This design requires *no external libraries or custom inference engine*, relying entirely on the database engine. Unlike prior in-database ML approaches that depend on external deep learning backend [3, 4, 17], TranSQL$^+$ uses portable SQL, enabling deployment on any standard database engines (e.g., PostgreSQL, DuckDB, Clickhouse) with minimal engineering effort. The method in this paper specifically targets CPU-only, memory-constrained hardware, where LLM inference typically serves a single request at a time, distinguishing it clearly from GPU-accelerated cloud-based inference

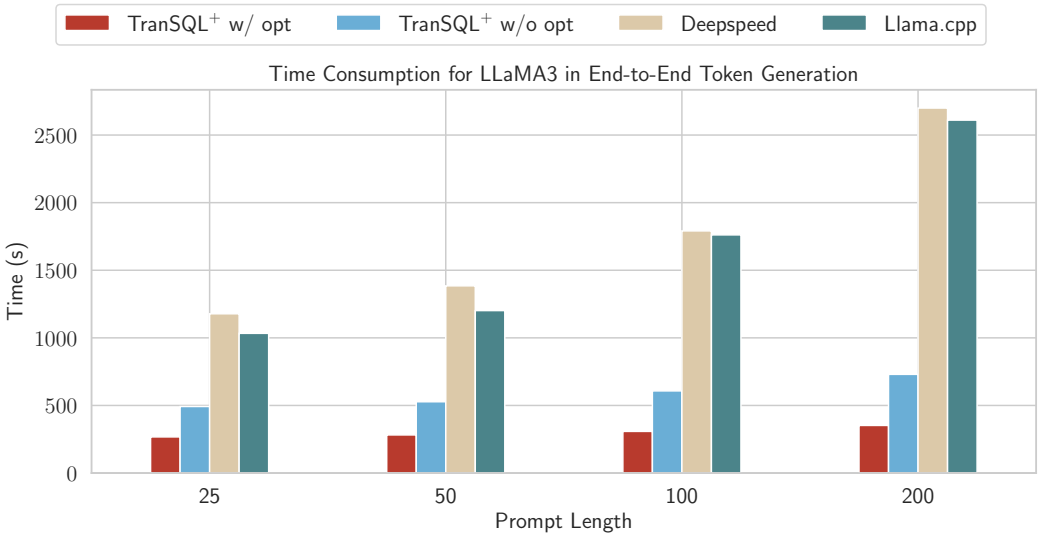


Fig. 1. End-to-end time consumption. The output token length is 50. TranSQL+ shows **10×** faster end-to-end (Prefill & Decoding) performance than production-ready frameworks.

services. Consequently, this paper's evaluations and optimizations are tailored explicitly to this scenario and are not directly comparable with GPU-oriented deep learning frameworks.

Contributions. This paper makes the following key contributions:

- We present *TranSQL+*, the first template-based SQL code generator capable of serving modern LLMs entirely within a relational database engine, requiring **no external dependencies**.
- We introduce ROW2COL and table fusion optimizations that leverage structural insights from matrix multiplication and the execution patterns of vectorized database engines to reduce intermediate result sizes and enhance parallelism. With these optimizations, TranSQL+ achieves a 2× speedup over the unoptimized SQL baseline.
- We evaluate TranSQL+ on two representative models—*Llama3 8B* and *DeepSeek MoE*—under a CPU-only setting (4 cores, 16GB RAM), showing up to **10**× speedups in end-to-end token generation compared to production-ready frameworks such as DeepSpeed Inference [37] and `Llama.cpp` [6] (see Fig. 1).

Our evaluation confirms that relational databases offer a practical, efficient, and highly portable backend for serving LLM inference in memory-constrained CPU-only environments overlooked by traditional frameworks.

This paper is organized as follows: Sec. 2 reviews background on LLM inference; Sec. 3 describes the design of TranSQL+ ; Sec. 4 details optimization techniques; Sec. 5 presents our experimental results; Sec. 6 discusses related work; and Sec. 7 concludes.

2 Background

LLMs are built on Transformer [45] architectures. During inference, the model receives a sequence of input tokens (e.g., words or subwords), encodes them into embeddings, and processes them through multiple layers to iteratively update these representations. The model then generates output tokens one at a time recursively.

This section introduces the fundamental computations involved in LLM inference, highlighting the core neural operators that underpin these models. Understanding these operators provides the foundation for our template-based approach to translating LLM inference into executable SQL queries. In addition to these micro-level operators, we also briefly present two representative model architectures—dense and mixture-of-experts (MoE)—to illustrate how structural differences can lead to varying performance characteristics when serving LLMs within relational databases.

2.1 LLM Inference Computation

Large language models (LLMs) like Llama3 [15] are implemented as deep transformer networks. Each Transformer layer applies two key sublayers – a self-attention mechanism [45] and a position-wise feed-forward network – each with a residual addition.

Self-Attention and GQA. The self-attention mechanism enables each token to attend to all others in a sequence. In a standard *multi-head attention* setup, the model computes separate *query* (Q), *key* (K), and *value* (V) projections for each attention head. Given an input token representation h, a single head computes:

$$Q = hW_Q, \quad K = hW_K, \quad V = hW_V,$$

where W_Q, W_K, and W_V are learned projection matrices. Given matrices Q, K, and V for all tokens, the attention output is computed as:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

where d_k is the dimensionality of each attention head.

To improve efficiency, recent LLMs such as Llama3 adopt a variant called *Grouped-Query Attention (GQA)* [8]. GQA partitions the H attention heads into G groups ($G < H$), where all heads in the same group share their key and value projections. For example, Llama1 [44] uses standard multi-head attention, but switches to GQA in the later versions [15] to reduce memory bandwidth consumption.

Formally, given queries $Q = [Q^{(1)}, \ldots, Q^{(H)}]$, GQA constructs only G distinct key-value pairs $K^{(1)}, \ldots, K^{(G)}$ and $V^{(1)}, \ldots, V^{(G)}$. Each head i is assigned to a group $g = f(i)$, and performs attention as:

$$\text{Attn}_i = \text{softmax}\left(\frac{Q^{(i)}(K^{(g)})^T}{\sqrt{d_k}}\right)V^{(g)}.$$

When $G = H$, this reduces to standard multi-head attention; when $G = 1$, it becomes *multi-query attention (MQA)* [41]—using shared K, V projections for all heads.

We adopt GQA in this work as it represents a more general and efficient formulation. Importantly, both GQA and standard multi-head attention rely on the same underlying neural operators (e.g., matrix multiplication, softmax), making them compatible with our SQL-based code generation framework.

SwiGLU Feed-Forward Activation. Following attention, each token passes through a position-wise feed-forward network (FFN). Llama3 uses the **SwiGLU** [42] activation—a gated, smooth nonlinearity formed by multiplying two linear projections (one passed through Swish). Unlike **ReLU**, which zeroes all negative inputs, and **GELU**, which applies a fixed Gaussian-shaped smooth gate, SwiGLU's learned gating improves gradient flow and expressiveness, yielding consistently better perplexity and convergence in large transformers. The SwiGLU activation is defined as:

$$\text{SwiGLU}(h) = (hW_1 + b_1) \odot \text{Swish}(hW_2 + b_2), \tag{2}$$

where $\odot$ is element-wise multiplication and $\text{Swish}(x) = x \cdot \sigma(x)$ with σ being the sigmoid function. The FFN output is then linearly projected back to the model dimension and added to the residual stream.

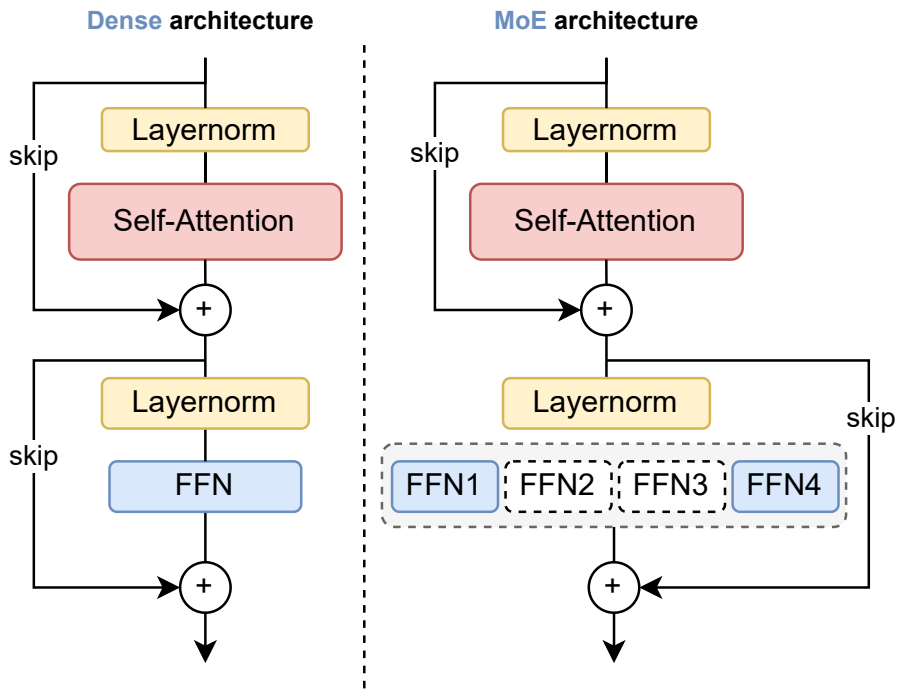


Fig. 2. Differences of dense models VS. MoE models

Summary. Each Llama Transformer layer consists of: (i) input normalization, (ii) multi-head or grouped-query attention, and (iii) a SwiGLU-activated feed-forward network, each has a residual connection to updated input embeddings. These operations form the core of modern LLM inference.

2.2 Prefill and Decoding

Token generation in LLM inference consists of two distinct phases: **prefill** and **decoding** [51]. During the prefill phase, the model processes the input context to compute the query (q), key (k), and value (v) vectors for all input tokens. This phase is the most resource-intensive, both in terms of computation and memory.

Once the first token is generated, the model enters the decoding phase, where new tokens are generated one at a time. In this phase, the attention vectors of previously generated tokens can be cached. As a result, only the q, k, and v vectors of the most recently generated token need to be computed, significantly reducing the computation cost compared to the prefill phase.

2.3 Mixture-of-Experts vs. Dense Transformers

An alternative approach to scale LLMs is the **Mixture-of-Experts (MoE)** architecture [16], exemplified by the recent LLMs, such as DeepSeek MoE [12] and QWen2MoE [48]. In a *dense* Transformer (e.g., Llama or GPT-4), every token activates the same set of parameters—typically a full feed-forward network—regardless of the input. In contrast, an *MoE* Transformer consists of multiple parallel feed-forward networks (experts), but only a subset of these experts is activated for any given token (see Fig. 2).

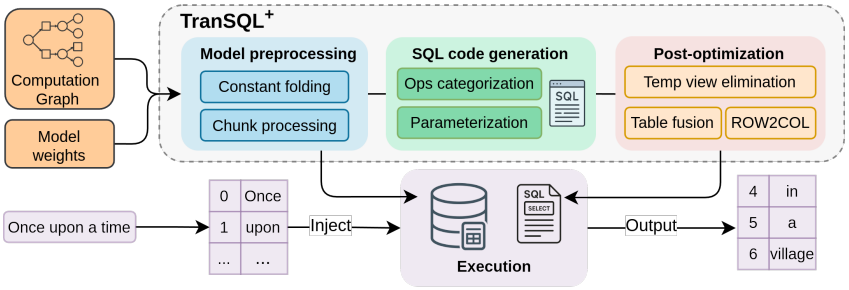


Fig. 3. An overview of the workflow in TranSQL+ .

Because only a small fraction of the parameters is active during each forward pass, only the relevant model weights need to be loaded into memory during inference. This makes MoE a compelling choice for deployment in resource-constrained environments, as MoE-based models can achieve similar task performance while running faster than their dense counterparts.

Moreover, MoE architectures are especially well-suited for database-based LLM serving during decoding. By partitioning weights and indexing by expert, databases can selectively load only the needed segments, leveraging native caching and query optimization to reduce I/O and manage memory efficiently. This synergy enables scalable, low-overhead inference. This collaboration between the MoE architecture and the inherent strengths of databases potentially facilitates fast, scalable, and memory-efficient inference.

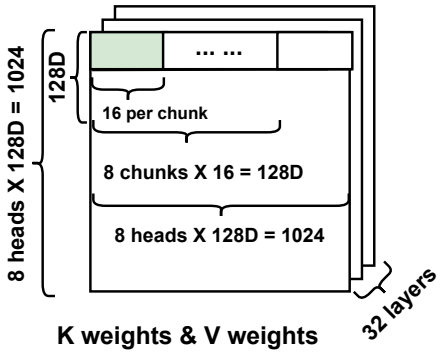
3 TranSQL+

TranSQL+ consists of three main components: *model preprocessing*, *template-based SQL code generation*, and *post-optimization* (see Fig. 3). To prepare model data for TranSQL+ , we first download the model's weights and configuration (e.g., embedding size, layer count) from repositories like HuggingFace[1].

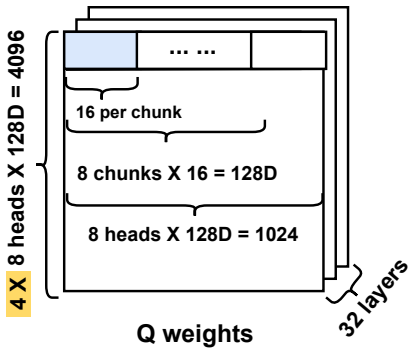
Model Preprocessing. The forward-pass computation graph is exported via ONNX [1] —an open, framework-agnostic neural network interchange format— and simplified using constant folding. Meanwhile, weights are loaded with PyTorch, chunked, mapped to a relational schema, and stored in columnar files (e.g., Parquet). If the input model follows a MoE architecture, TranSQL+ additionally builds indexes on expert identifiers to efficiently retrieve only the activated expert weights during inference. The database files are larger than the raw model tensor files because a database stores additional metadata (schemas, catalogs, pages, indexes, free-space maps, etc.) rather than just contiguous value arrays. Concretely, Llama3 8B grows from 16.1GB (raw) to 21.3GB in the database, and DeepSeek MoE from 32.8GB to 48.6GB, including 0.7GB of indexes for experts. Llama3 uses no index: as a dense model, nearly all weights participate in computation, yielding low selectivity and little benefit from indexing. The preprocessing runs once on a server-class machine when integrating a new model; the resulting SQL queries and data can then be transferred to target devices.

SQL Code Generation. Given the computation graph of an LLM inference pass, TranSQL+ extracts all neural operators and maps each to a corresponding SQL template. The generator fills in operator-specific parameters (like input/output shapes and aggregation dimensions), translating each operator into a standalone SQL query. These queries are then connected according to the computation graph, forming a directed acyclic graph (DAG) of SQL queries.

[1]https://huggingface.co/



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...>

Fig. 4. Illustration of slicing K, V, Q weights into chunks.

Post-optimization. TranSQL+ applies a **ROW2COL** transformation to matrix-vector and matrix-matrix multiplication queries. This reduces the cardinality of intermediate results, thereby enhancing `JOIN` and `GROUP BY` efficiency. Additionally, consecutive queries are merged into common table expressions (CTEs) to minimize intermediate data materialization and I/O overhead, with natural segmentation occurring at embedding-generation boundaries. Then the final output is generated as a SQL script stored in the local file system.

Execution. When a prompt arrives, an external tokenizer extracts the input tokens and produces an SQL script that inserts those tokens and their corresponding position IDs into the database. Following that, the database then executes sql script generated by TranSQL+ recursively, generating tokens one at a time. Each generated token is stored in an output table, which maintains the full decoding history and can be reused in subsequent steps.

3.1 Model Preprocessing

3.1.1 Chunk-Based Representation. To execute neural operators via SQL queries, we first convert model weight matrices into relational tables. We represent matrices in a *chunked* format, as explored in previous work for in-database LA [22, 34]. Matrices are split into smaller blocks, indexed by row and column. However, because databases rarely offer native matrix support but often support vector operations (e.g., dot products and element-wise arithmetic [36, 40]), we partition matrices into vectors for broader compatibility. Fig. 4 illustrates an example of this chunk-based representation.

Specifically, for a large matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, we split each row into $\lfloor \frac{n}{chunk\_size} \rfloor$ chunks:

$$\mathbf{w}_i = \left[\mathbf{w}_i^{(0)}, \mathbf{w}_i^{(1)}, \ldots, \mathbf{w}_i^{(C)}\right],$$

where each chunk $\mathbf{w}_i^{(c)}$ has a fixed dimension determined by a *chunk size* hyperparameter. In the database table, each row thus corresponds to:

$$(i,\ c,\ \mathbf{w}_i^{(c)}),$$

for $0 \le i < m$ and $0 \le c < \lfloor \frac{n}{chunk\_size} \rfloor$. This chunking strategy provides flexibility by allowing trade-offs between the number of rows and the workload per thread when performing vector operations.

Algorithm 1 Convert Matrix To Relational Tables

```
procedure ConvertMatrixToChunks(matrix, chunk_size)
    M ← number of rows in matrix
    chunkTable ← ∅
    for i = 0 to M − 1 do
        row ← matrix[i]
        N ← length of row
        for c = 0 to N − 1 step chunk_size do
            end ← min(c + chunk_size, N)
            chunk ← slice of row from index c to end − 1
            add tuple (row_index : i, chunk_index : c, vector : chunk) to chunkTable
        end for
    end for
    return chunkTable
end procedure
```

Algorithm 1 outlines the procedure for converting weight matrices into chunked representations and producing a tuple list suitable for database import. For traditional databases without array support, TranSQL$^+$ falls back to a chunk size of 1, storing all matrices in a `(row_id, col_id, value)` schema. Vector operations are rewritten as scalar operations combined with `GROUP BY`–aggregation patterns in the generated SQL.

Notably, when a matrix appears as the *second* operand of a multiplication (e.g., $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$), TranSQL$^+$ physically transposes B during preprocessing—storing its column vectors as row vectors—so dot products can be computed directly at inference time without an extra transpose. This alters only the physical layout, not the original computation graph. Our chunking scheme therefore assumes no *algebraic* transposition is required at runtime—an assumption that holds for the transformer models studied here (Llama3 and DeepSeek MoE). We discuss cases that require transposition or re-chunking in Sec. 3.2.2.

SQL Implementation of Matrix Multiplication. Within this chunked table structure, we emulate the summation $\sum_c a_{ic}b_{cj}$ by joining the chunked rows of the two matrices on their common index k. The relational engine then multiplies corresponding entries and aggregates the products to form each entry of the resulting matrix $\mathbf{W}$. Formally, the SQL implementation follows a pattern like:

```
SELECT
    A.i AS i, B.j AS j,
    SUM(A.w_i^(c) · B.w_j^(c)) AS w_ij
FROM A JOIN B ON A.c = B.c GROUP BY A.i, B.j;
```

F Here, A.$\mathbf{w}_i^{(c)}$ and B.$\mathbf{w}_j^{(c)}$ would each represent chunked slices of the matrices, typically stored as arrays or separate columns. The result is then reorganized (if needed) to match the output dimension $m \times n$.

This SQL-based matrix multiplication serves as a foundation for implementing other operations (e.g., element-wise activations, softmax) directly in the relational engine, enabling end-to-end inference within a database system.

3.1.2 Constant Folding. Constants are commonly present in the computation graphs of LLMs, typically serving as parameters for other neural operators. They can take the form of scalars (e.g., token length) or matrices (e.g., weights). TranSQL$^+$ performs *constant folding*: constant tensors and

Table 1. Templates for primary operator categories and examples.

Operator Category	Operator	Template	SQL Query
Matrix multiplication	AB	$R_3 \leftarrow \Pi_{\mathcal{G},S,\mathrm{DOT}(c_1,c_2)}(R_1 \bowtie_S R_2)$ $\Pi_{\mathcal{G},\mathrm{SUM}(c_3)}(\gamma_{\mathcal{G}}(R_3))$	SELECT A.m, B.q, SUM(DOT(A.chunk, B.chunk)) FROM A JOIN B ON A.n = B.p GROUP BY A.m, B.q
Element-wise function	Sigmoid(A)	$\Pi_{\mathcal{F},f(c_1)}(R_1)$	SELECT A.m, A.n, 1/(1+exp(-A.chunk)) FROM A
Element-wise arithmetic	$A + B$	$\Pi_{S,f(c_1,c_2)}(R_1 \bowtie_S R_2)$	SELECT A.m, A.n, A.chunk + B.chunk FROM A JOIN B ON A.m = B.p AND A.n = B.q
Reshape	A.flatten()	$\Pi_{\mathcal{G},f(S),c_1}(R_1)$	SELECT A.m*M+A.n, A.chunk FROM A
Normalization	Softmax(A)	$R2 \leftarrow \Pi_{\mathcal{F},\mathcal{G},\mathrm{agg}(f(c_1))}(R_1)$ $R3 \leftarrow \Pi_{\mathcal{G},\mathrm{agg}(c_2)}\gamma_{\mathcal{G}}(R_2)$ $\Pi_{\mathcal{F},\mathcal{G},g(c_2,c_3)}(R_2 \bowtie_S R_3)$	WITH exp_sum AS (SELECT A.m, SUM(SUM(exp(A.chunk))) AS summation FROM A GROUP BY A.m) SELECT A.m, A.n, exp(A.chunk)/summation FROM A JOIN exp_sum ON A.m = exp_sum.m

any downstream linear or scalar operations on them are precomputed to avoid redundant work at inference time. In general, for a constant A in an expression of the form $\mathrm{Op}(AB) = \mathrm{Op}(A)\,B$, we evaluate $\mathrm{Op}(A)$ offline. For instance, the attention scaling factor $\frac{1}{\sqrt{d_k}}$ in Eq. 1 can be absorbed into the weight W_Q, eliminating the scalar division during inference. This optimization avoids unnecessary computation during SQL execution and simplifies the resulting query plan.

3.2 Template-based SQL Code Generation

To translate LLM inference computations into executable SQL, we adopt a *template-based SQL code generation* approach. Each neural operator maps to a predefined SQL template whose placeholders are instantiated with model-specific parameters—e.g., attention-head group size (GQA) and token/embedding dimensionality—enabling TranSQL+ to flexibly support diverse LLM architectures. This method builds on prior work in structured SQL generation [38, 47], which shows that templates are expressive enough to cover common LA operators.

A major strength of this approach is its *extensibility*: new operators can be supported by simply adding templates. As demonstrated in our evaluation, this flexibility allows TranSQL+ to handle both transformer components and convolution kernels.

Table 1 summarizes the neural operator categories alongside representative SQL queries. In this section, we focus on the template design for the most important and complex operator categories: matrix multiplication, element-wise functions, and normalization. Additional details for reshape and element-wise arithmetic categories are provided in Appendix.

3.2.1 Parameterize neural operators. To match neural operators to their corresponding SQL templates, TranSQL+ first identifies the minimal set of parameters required to instantiate a valid SQL query. For each operator, we record its operand relations, operator type, and index structure (shared dimension vs. free dimensions), then instantiate a parameterized template that compiles to SQL. (i) *Shared dimensions* are indices along which operands interact (multiply, sum, compare). In SQL, they become join keys and aggregation attributes. (ii) *Free dimensions* do not participate in these interactions; in SQL, they appear as grouping or projection columns carried to the output. For example, in $C_{ij} = \sum_k A_{ik}B_{kj}$, k is shared (it mediates interaction and is eliminated), while i and j are free, becoming the row and column indices of C.

During preprocessing, TranSQL+ categorizes all neural operators in the inference computation graph into the following types: (i) element-wise functions (e.g., `Sigmoid`, `SiLU`), (ii) element-wise arithmetic (e.g., vector addition, element-wise multiplication), (iii) matrix–matrix and matrix–vector multiplication, (iv) shape manipulation (e.g., `view`, `reshape`), and (v) normalization-related operations (e.g., `RMSNorm`, `Softmax`). After categorization, raw tensor operands' shared dimensions are

transformed into chunk-based representations and subsequently converted into relational tables, as detailed in Sec. 3.1.

Formally, a neural operator acting on p operands is represented as:

$$\mathrm{OP}_{\texttt{attr}}\Big(\{R_1, \ldots, R_p\},\ \mathcal{F},\ \mathcal{S},\ \mathcal{G}\Big),$$

where $\mathcal{F}$ denotes the set of *free dimensions* retained in the output for each operand R_i, and $\mathcal{G}$ represents the set of dimensions to be used in `GROUP BY` clauses in the SQL template. $\mathcal{S}$ encodes the mappings of *shared dimensions* between operands, which are further translated into SQL join keys. The subscript `attr` indicates a set of operator-specific attributes that guide the final projection logic, such as function types, reshaping rules, or normalization strategies.

Matrix-Matrix Multiplication. Matrix multiplication $\mathbf{X} = \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{m\times r}$ and $\mathbf{B} \in \mathbb{R}^{r\times n}$, can be calculated through:

$$x_{ij} = \sum_{k=0}^{\lfloor \frac{r}{c} \rfloor} \mathbf{a}_{ik} \cdot \mathbf{b}_{kj}, \quad \text{for } i \in [0, m), j \in [0, n),$$

where free dimensions m and n and shared dimension r (subdivided into chunks by c) are identified. Matrix multiplication is parameterized as:

$$\mathrm{Matmul}\Big(\{A, B\},\ \varnothing,\ \{(r_A, r_B)\},\ \{m_A, n_B\}\Big), \tag{3}$$

where A and B are the input matrices, $\{m_A, n_B\}$ are the *group-by dimensions* corresponding to the output row and column indices, and $\{(r_A, r_B)\}$ represents the *shared dimensions* over which the summation is performed. When input data is imported into the database, the right-hand operand in matrix multiplication is transposed to convert its column vectors into row vectors, aligning with how dot products are computed in databases. Note that the symbolic notation used here abstracts away from the actual physical relational schema.

Since the summation in matrix multiplication translates to a `GROUP BY + SUM` operation in SQL, both m_A and n_B are included in the set $\mathcal{G}$ to ensure the correct grouping of partial products in the SQL query.

Element-wise functions. Element-wise functions apply a transformation independently to each element of matrices using a function f. A representative class of such operations includes activation functions such as `Sigmoid`, `ReLU`, and `SiLU`. Since these operations do not involve any interactions across elements—i.e., no joins or aggregations are required—all dimensions are considered *free dimensions*. For example, Sigmoid(A), where $A \in \mathbb{R}^{m\times n}$, can be parameterized as:

$$\mathrm{Sigmoid}\Big(\{A\},\ \{m_A, n_A\},\ \varnothing,\ \varnothing\Big).$$

Because Sigmoid function applies to elements in matrix A independently, the shared dimension here is an empty set.

Normalizations. Normalization-like operations are ubiquitous in LLM architectures. These operators typically follow a two-step pattern: (i) apply a function to each element, (ii) apply aggregation in chunks as well as over a shared dimension, and (iii) apply a function to each element and aggregated values over the shared dimension. This general pattern not only includes statistical normalizations (e.g., `layernorm`), but also functions like `softmax`.

To capture the structure of normalization operations, we parameterize them using three components: an element-wise function f, an aggregation function *agg*, and a final element-wise post-processing function g. These functions define the sequence of transformations in the SQL template: applying f to the input, aggregating with *agg* over the shared, and then applying g to normalize

each element using the aggregated result. We illustrate this with two representative examples: `softmax` and `layernorm`.

`Softmax` is typically applied row-wise, transforming a vector $\mathbf{a}_i$ from a matrix $A \in \mathbb{R}^{m \times n}$ as:

$$\text{Softmax}(\mathbf{a}_i) = \frac{\exp(a_{ij})}{\sum_{j=0}^{n-1} \exp(a_{ij})}.$$

This operation first applies exp element-wise, computes a sum over the shared dimension (i.e., across each row), and finally normalizes each element by dividing by the corresponding aggregated sum.

We express this using the following parameterization:

$$\text{Normalize}_{\text{exp, SUM, div}}\left(A,\ \{n_A\},\ \{(m_A, m_A)\},\ \{m_A\}\right),$$

where exp is the first element-wise function, `SUM` is the aggregation over $\{n_A\}$, and `div` performs the final normalization step.

Similarly, `layernorm` can be represented in the same framework by using the `identity` function as the first step (i.e., $f = \texttt{id}$), which leaves the input unchanged before aggregation.

Furthermore, building on the `layernorm` pattern, we can perform global aggregation across each row of the input matrix by modifying the parameterization: specifically, by setting the final function g to the `identity` function and assigning all dimensions as shared dimensions. For example, this allows us to compute the maximum value in each row through a single aggregation step. These variants highlight the expressiveness and flexibility of our parameterization framework in capturing a wide range of normalization and reduction behaviors within a unified abstraction.

3.2.2 Special Cases. Some neural operators fall outside the five core categories discussed earlier. While these cases are relatively infrequent, TranSQL+ handles them individually due to their unique computational semantics.

Lookup Table. Lookup tables are commonly used in the input layer to retrieve token embeddings and in the output layer to map final activations to vocabulary tokens. These operations can be efficiently implemented using a simple equi-join on token strings or token IDs. For instance, to retrieve embeddings for a sequence of tokens stored in an `input_token_table`, we can join it with an `embedding_table` that maps tokens to their corresponding vector representations:

```
SELECT
    input.token_id, embed.chunk_id, embed.chunk
FROM input_token_table AS input
JOIN embedding_table AS embed
ON input.token = embed.token
```

Here, `row_id` maintains the original token order, and the join retrieves the matching embedding vector for each token. This pattern supports both input embedding lookup and final output decoding, and integrates naturally into the relational execution framework.

Transpose. The transpose operation is non-trivial under a chunk-based representation, as it requires unnesting and reassembling all values with swapped dimension indices. In the models discussed here, no runtime transposition is needed. Weight matrices are static, and intermediate matrices are consumed only by specific, fixed operations. Thus, if a transposition appears in the original computation graph, we can instead rewrite the operation by pre-transposing the corresponding weight matrix. For example, consider a sequence of matrix multiplications:

$$X = A \cdot (BC^\top)^\top,$$

where A, B, and C are input matrices. A naive evaluation would compute $BC^\top$, then transpose the intermediate result, and finally multiply with A. However, we can algebraically rewrite this as:

$$X^\top = BC^\top A^\top,$$

which avoids the intermediate transpose. To enable this transformation, we store $A^\top$ in the database during model import instead of A, effectively bypassing the need for transposition during inference.

If intermediate transposes are required for other model types, TranSQL$^+$ can incorporate them purely in SQL. Starting from a chunked layout `(row_id, col_chunk, vec)`, we (i) unnest each vector into tuples `(row_id, col_id, value)` with `col_id = col_chunk * chunk_size + offset`; (ii) swap indices to `(col_id, row_id, value)`; and (iii) re-aggregate by `col_id` into chunked vectors, yielding the transposed table. This incurs some overhead but does not compromise the completeness or correctness of our approach.

3.2.3 SQL code generation. Using the parameterization results, we populate predefined SQL templates to generate queries, each forming a temporary view with a unique identifier. Most of these views are later merged during post-optimization via `WITH`-clause rewriting, as detailed in the next section.

In Table 1, we follow standard relational algebra notations: Π_* denotes a *projection*, where $*$ indicates either a set of attributes or functions applied to them; $\bowtie_*$ represents a *join*, with $*$ specifying an equi-join condition (or a cross join if omitted); and γ denotes a *group-by* operation, typically followed by a projection applying an aggregation function.

We assume input tensors with dimensions $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$. In their chunk-based relational representation, c_A and c_B refer to the chunk identifiers for A and B, respectively. The functions applied to chunks are vector operations supported by databases.

The five core categories and special cases in our framework cover the neural operators used in transformer-based LLMs (e.g., LLaMA, QWen, DeepSeek MoE). With the templates presented here, we already support both dense and MoE models in this family. While not exhaustive, SQL's expressiveness and our modular template design allow easy integration of new operators through additional templates and parameter mappings. By extending our templates and implementing parameterization accordingly, TranSQL$^+$ can support broader model families. For convolutional networks, 2D convolutions can be rewritten via the common *im2col* trick into matrix multiplications and thus reuse our MatMul template; pooling layers can be expressed by generating window coordinates (via row/column indices) followed by a `GROUP BY`–`MAX` (or `AVG`) aggregation, requiring only minor template additions. The main change is the input format (image tensors instead of token sequences); once ingested, the workflow in Fig. 3 remains unchanged. Crucially, these adaptations are one-time template extensions—no modifications to the execution environment—demonstrating TranSQL$^+$'s extensibility and portability.

4 Post-optimization

Although the SQL templates translate neural operators into view creating queries, end-to-end performance can be suboptimal: repeated materialization incurs heavy I/O, and large intermediates, especially from matrix multiplication, slow `JOIN`/`GROUP BY` and constrain parallelism. We address this with three optimizations that reduce I/O workload and expose more parallel work on vectorized engines: *temporary view elimination*, *table fusion*, and *ROW2COL* pivoting. Together, these changes better exploit the planner and execution engine and substantially improve performance.

4.1 Temporary View Elimination

After generating SQL templates for individual neural operators, the next step is to assemble these templates following the original computation graph. This process initially produces many SQL subqueries and temporary views, which can cause significant I/O overhead. To address this, we leverage Common Table Expressions (CTEs) to merge multiple subqueries into a single, unified SQL statement. The optimization procedure is summarized in Algorithm 2.

The computation graph is first topologically sorted (line 4) to ensure correct execution order. Each node is then visited to determine whether it can be merged into the preceding CTE block. If a node is identified as a *critical node*, the currently accumulated nodes are materialized into a table (lines 5–12); otherwise, the node is absorbed into the ongoing CTE expression. A critical node is one whose output (i) is consumed by multiple downstream operators or (ii) is subsequently modified by one or more operators. To avoid recomputation, we materialize such outputs as tables. In transformer-based LLMs, critical nodes fall into two categories:

Key and Value Vectors of Attention Heads: These vectors are essential for constructing the key–value cache [35] used in subsequent decoding steps. Materializing them allows efficient reuse during generation without recomputing the attention mechanism.
Embeddings for Residual Connections: In models with residual connections, the original input embeddings must be preserved for later addition to the updated embeddings, as illustrated in Fig. 2 (marked as *skip*). Materializing these embeddings ensures that the residual paths are correctly maintained during inference.

In the final step, any remaining unvisited nodes are folded into the last CTE, completing the SQL script; this optimization cuts the number of materialized intermediates per layer for Llama3 8B from 38 to 7, substantially reducing I/O from repeated materialization and scans.

4.2 Table Fusion

In LLM inference, the *query*, *key*, and *value* vectors are essential intermediate results used to compute attention scores for each token. These vectors are obtained by projecting the input embeddings through their respective weight matrices:

$$Q = \text{embedding} \times W_Q,\ K = \text{embedding} \times W_K,\ V = \text{embedding} \times W_V,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are the projection weights.

Although computed independently in the model's computation graph, these projections share the same input and produce outputs with identical schema and cardinality. Naively materializing them separately results in redundant scans of the embedding table and generates three temporary output tables, breaking the continuity of the CTE pipeline and increasing I/O overhead. Moreover, since these vectors are consumed together in a single attention layer, querying them independently prevents the query optimizer from performing fine-grained parallelization, as it can only schedule at the table level.

To address this inefficiency, we adopt a fusion-based optimization by concatenating the projection weights into a single extended matrix:

$$[Q \parallel K \parallel V] = \text{embedding} \times W_{QKV}, \quad \text{where } W_{QKV} \in \mathbb{R}^{d_{\text{model}} \times 3d_k}.$$

In TranSQL+ , this is implemented as a table fusion optimization. The weight tables for W_Q, W_K, and W_V are vertically merged using `UNION ALL`, with a flag column distinguishing the projection type. During SQL generation, this flag becomes part of the shared dimensions, allowing all three vectors to be computed in a single SQL query. Figure 5 illustrates this optimization.

Algorithm 2 Temporary View Elimination

```
1: procedure OptimizeCTEQueries(nodes, graph)
2:     finalSQL ← []                                   ▷ List of final SQL statements
3:     cteChain ← []                                   ▷ Cache of SQL fragments for merging via CTEs
4:     topoNodes ← TopologicalSort(graph)              ▷ Obtain nodes in topological order
5:     for all node in topoNodes do
6:         if IsCriticalNode(node) then                ▷ Determine if node requires materialization
7:             if cteChain ≠ [] then
8:                 cteQuery ← MergeCTEs(cteChain)      ▷ Merge non-critical nodes into one CTE query
9:                 Append cteQuery to finalSQL
10:                Clear cteChain
11:            end if
12:            Append node.SQL_query to finalSQL       ▷ Materialize critical node separately
13:        else
14:            Append node.SQL_query to cteChain       ▷ Accumulate non-critical node query
15:        end if
16:    end for
17:    if cteChain ≠ [] then
18:        cteQuery ← MergeCTEs(cteChain)
19:        Append cteQuery to finalSQL
20:    end if
21:    return CombineSQLStatements(finalSQL)           ▷ Combine all SQL statements into a final script
22: end procedure
```

Merging Q/K/V weight tables into one single relation lets us scan the weights once and compute all three matrix multiplications in a single MatMul SQL query, distinguished only by different `GROUP BY` keys. Since vectorized engines load contiguous column chunks into SIMD registers, a unified table lets the executor stream Q/K/V columns together, avoiding cache flushes and repeated scans while exposing finer-grained parallelism (each column block can be pipelined across cores). The same idea applies to SwiGLU feed-forward layers: the two parallel projections can be fused into one table, enabling a single pass that computes both branches and eliminating redundant joins and I/O.

4.3 ROW2COL Pivoting

Although SQL is expressive for neural operators, executing them directly in relational engines can be inefficient because join-based alignment inflates intermediate results, forces key materialization, and limits intra-operator parallelism. In matrix multiplication $X = AB$, tensor frameworks (NumPy [19], PyTorch [21]) access rows and columns by array index, whereas relational systems align vectors via scans and equi-joins—as illustrated by the matrix-multiplication example in Sec. 3.1.1—incurring additional overhead.

Unlike tensor frameworks that operate on in-memory dense matrices and traditional *row-oriented* RDBMSs, modern analytical engines (e.g., DuckDB, ClickHouse, Greenplum) use columnar storage to exploit parallelism across independent columns and to improve cache locality on contiguous column values. Building on the chunked formulation of matrix multiplication—where chunk k of A pairs one-to-one with chunk k of B—we introduce ROW2COL, a logical schema rewrite that

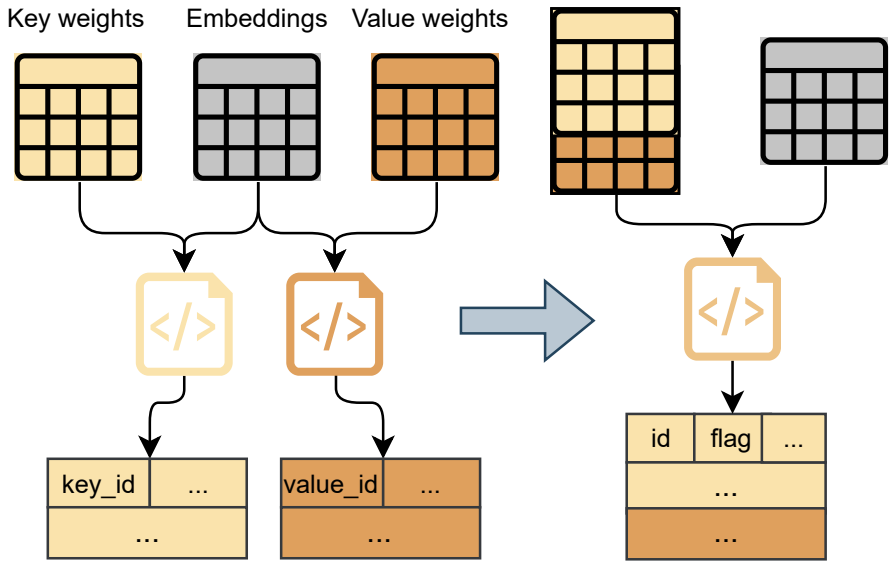


Fig. 5. Merge computation of key and value vectors into a single CTE.

pivots chunk rows into columns. ROW2COL reduces join cardinality and scanned rows and exposes contiguous, independent column slices for parallel processing. The transformation uses `PIVOT` where available (e.g., SQL Server, DuckDB) or conditional aggregation with `CASE` otherwise.

For simplicity, we use DuckDB built-in `PIVOT` syntax as an example. Assume A and B are chunked into 64 vectors. The original schema is `(row_id, chunk_id, vec)`. ROW2COL pivots this to `(row_id, chunk0, . . . , chunk63)`, so each chunk becomes a dedicated column. Based on the new schema, ROW2COL rewrites the matrix multiplication query in Sec. 3.1.1 as:

```
WITH c0 AS (SELECT A.row_id, B.row_id,
        DOT(A.chunk0, B.chunk0) AS v0
    FROM A_pivot CROSS JOIN B_pivot
    ORDER BY A.row_id, B.row_id),
    ...
c63 AS (SELECT A.row_id, B.row_id,
        DOT(A.chunk63, B.chunk63) AS v63
    FROM A_pivot CROSS JOIN B_pivot
    ORDER BY A.row_id, B.row_id)
SELECT A.row_id AS i, B.row_id AS j,
    v0+...+v63 AS value_ij
FROM c0 POSITIONAL JOIN c1
```

This optimized query yields multiple benefits: each CTE subquery computes the dot product for a single chunk pair, enabling independent, parallel execution; the final step replaces a costly `Equi-Join` with a `POSITIONAL JOIN` that aligns rows by order, eliminating explicit key matching. Overall, *ROW2COL* (i) cuts the number of source rows and the cardinality of intermediate joins, accelerating scans and join processing; (ii) exploits modern vectorized engines [20, 36, 53] by treating columns as contiguous arrays, reducing function-call overhead and improving cache

locality; and (iii) exposes column-aligned chunks that can be processed in parallel, increasing opportunities for the optimizer to schedule concurrent work.

Nevertheless, ROW2COL is not a silver bullet. Although DuckDB [36] and ClickHouse [40] support `POSITIONAL JOIN`, it is non-standard and often unavailable elsewhere, so fallbacks such as `SORT–MERGE JOIN` are needed; these can perform comparably when subqueries are consistently pre-ordered. Pivoting also changes the schema and can introduce extra materialization, and in some cases this overhead outweighs the reduced join costs. Very wide pivots can strain planning and execution, especially once the column count exceeds effective parallelism. We empirically characterize these trade-offs in Section 5.3 under varying hardware settings.

We therefore apply ROW2COL conservatively: only when the operator is chunk-parallel with a moderate number of chunks (e.g., embeddings or K/V tables), the widened schema remains within the engine's practical limits (vector width, planner constraints), and the anticipated drop in join cardinality outweighs the pivoting cost (heuristically estimated from chunk count, and vector width). We skip ROW2COL for extremely wide pivots or very large chunks (CPU saturation). A full cost model would weigh pivot cost, expected cardinality reduction, width relative to core count, and cache pressure. Designing such a comprehensive cost model requires substantial effort, which we leave for future work.

5 Evaluation

In this section, we comprehensively evaluate the performance, efficiency, and generality of TranSQL$^+$ as a lightweight SQL-based inference engine for large language models and deep learning operators. Our evaluation is designed to answer the following questions:

Q1: How does TranSQL$^+$ compare to state-of-the-art inference frameworks regarding latency under resource-constrained settings?

Q2: What is the contribution of post-optimization techniques to overall performance?

Q3: How does memory capacity impact execution efficiency?

Q4: Can the same SQL-based framework be extended to support other types of neural operators, such as convolutions?

We evaluate both dense and MoE LLMs using real-world models (Llama3 and DeepSeek MoE), run experiments on low-memory hardware, and compare against two widely-used inference frameworks—DeepSpeed and Llama.cpp. Additionally, we benchmark TranSQL$^+$ against DL2SQL with convolution kernels to demonstrate extensibility.

5.1 Experimental Setup

5.1.1 Models. We evaluate TranSQL$^+$ on two representative LLMs: Llama3 8B (16.1GB), a dense transformer with GQA and SwiGLU, and DeepSeek MoE [12] (32.8GB), a mixture-of-experts model with 6 expert activations for each token. These models reflect the core architectural patterns of many popular LLMs (e.g., QWen [48], Mixtral [25]) and cover the key neural operators addressed by our SQL templates. The models used here are unquantized full-precision, as relational databases lack native support for quantized arithmetic. Compression methods are orthogonal to this research and can be applied separately.

Prompt tokens are pre-generated, and each run at a given length uses the same prompt. We restart the database before every run to clear caches. Prompt lengths follow the LMSys-Chat [50] average of 69.3 tokens, so we test 25–200 tokens to cover typical conversations. Although very long prompts are practical in some settings, they are relatively uncommon on resource-constrained devices acting as personal assistants. Evaluating up to 200 tokens is sufficient to reveal TranSQL$^+$'s advantages and limitations.

5.1.2 Hardware and Database. We conduct experiments on an AWS `c7.2xlarge` instance with 4 physical CPU cores and 16GB RAM—comparable to typical personal devices. Our focus is on evaluating TranSQL+ under memory-constrained conditions, not on edge-specific CPU architectures; since all baselines run on the same hardware, comparisons remain fair. To simulate tighter memory limits, we also evaluate performance under reduced RAM in Sec. 5.4.

We use DuckDB [36] as our backend, a lightweight analytical database supporting out-of-core execution, vectorized processing, and array operations—key features for expressing neural computations in SQL. To demonstrate TranSQL+ 's compatibility with other databases, we also evaluate its performance on ClickHouse [40].

5.1.3 Baselines. We compare TranSQL+ with two production-grade inference frameworks: DeepSpeed Inference and Llama.cpp.
(I) *DeepSpeed* is a transformer inference engine developed by Microsoft that includes support for tensor parallelism, kernel fusion, and NVMe-based weight offloading. It is widely used in production-level deployments.
(II) *Llama.cpp* is a lightweight LLM serving framework. It enables out-of-core execution via memory-mapped weights and is especially popular for edge deployments and personal devices.

Several research efforts have explored model offloading to disk; however, many of these approaches either require an additional training step to summarize model weights—effectively producing a new model [9, 46]—or are specifically tailored for GPU-based execution and memory management, limiting their generalizability [43]. Since our goal is to explore a broadly applicable method for deploying LLMs on personal or edge devices, these approaches are not directly comparable to TranSQL+ . Instead, we select `DeepSpeed` and `Llama.cpp` as baselines due to their wide adoption and contrasting design goals: `DeepSpeed` targets both high-performance training and inference, while `Llama.cpp` focuses on inference on various hardware settings. Crucially, both support inference with model sizes that exceed physical memory, aligning well with our deployment setting.

5.1.4 Metrics. We report performance using two key metrics that correspond to the typical stages of LLM inference: prefill latency and decoding latency. Prefill latency measures the time required to process the input prompt and produce the first output token. This stage is the most computationally intensive, as it involves full attention and feed-forward passes over the entire prompt. Decoding latency measures the average time taken to generate each subsequent token, assuming cached key and value vectors.

5.2 Overall Latency Evaluation (Q1)

We evaluate the end-to-end inference performance of TranSQL+ *with all post-optimizations* using two large language models—Llama3 8B and DeepSeek MoE—across a range of input prompt lengths. We report the latency for both the prefill phase (i.e., time to generate the first token) and the decoding phase (i.e., average time per generated token after the first).

Prefill Latency. Fig. 6 summarizes the prefill latency. In the Llama3 and DeepSeek MoE models, TranSQL+ on DuckDB and ClickHouse consistently achieves lower prefill latency than both deep learning serving frameworks across all prompt lengths. Moreover, TranSQL+ on DuckDB is faster than on ClickHouse, indicating that performance depends on the underlying engine. As prompt lengths increase from 25 to 200 tokens, the latency of all baselines grows, but TranSQL+ demonstrates significantly better scalability. This is especially evident in the MoE model, where the benefits of our query optimizations—such as `ROW2COL` pivoting and selective materialization—are amplified due to the model's partially activated pattern.

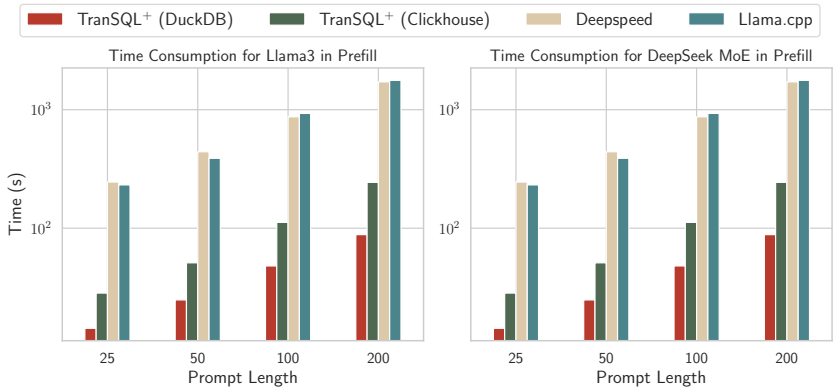


Fig. 6. Prefill latency with varying prompt length. TranSQL+ gains up to 20x speedups compared to Deepspeed and Llama.cpp.

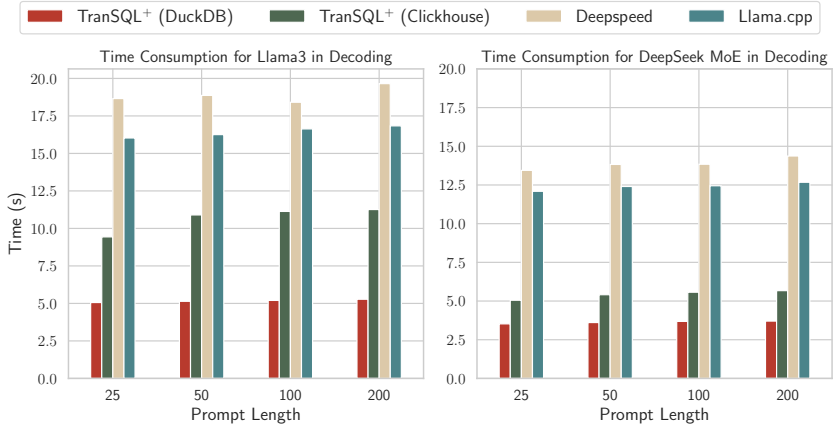


Fig. 7. Decoding latency with varying prompt length. TranSQL+ gains up to 4x speedups compared to Deepspeed and Llama.cpp.

Decoding Latency. During decoding, where only one token is generated at a time, the performance gap between methods narrows but remains consistent, as shown in Fig. 7. TranSQL+ continues to outperform all other approaches on both models. The gap between DuckDB and Clickhouse also narrows due to less computation and I/O workload. In the dense Llama3 model, decoding times remain relatively stable across prompt lengths.

Effectiveness of Index for DeepSeek MoE. Database indices accelerate MoE inference by exploiting low selectivity—only a small subset of experts is active per token. Fig. 8 shows the speedup from indexing the expert layer in DeepSeek MoE. During decoding, indexing yields a nearly stable 1.78× speedup, since only 6 of 64 experts are queried, reducing the table scan time. In the prefill stage, longer prompts activate more experts, causing the computation to resemble dense inference (as in Llama3). Consequently, by 200 tokens, the index provides negligible benefit.

Performance on GPUs. For reference, we also ran Llama3 8B on a single A100 GPU (with sufficient memory to hold all weights) using PyTorch. Prefill latency for a 200-token prompt was

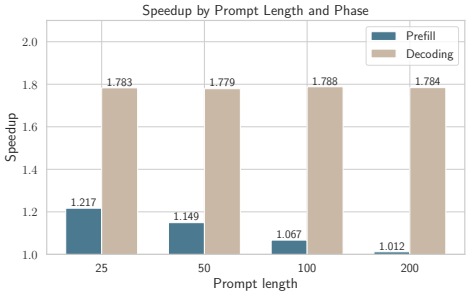


Fig. 8. Speedup of Prefill with index vs. without index on Experts for DeepSeek MoE

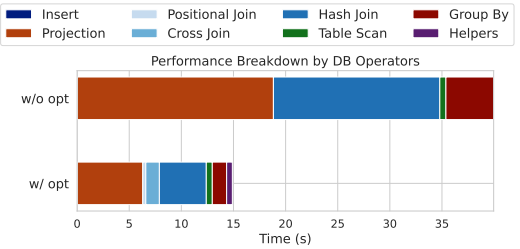


Fig. 9. Join time drops by over 60% because the optimizations reduce intermediate cardinality. "w/ opt" denotes TranSQL+ with all three optimizations; "w/o opt" denotes TranSQL+ without any optimizations.

approximately 60ms, and per-token decoding latency was approximately 10ms—far below any baselines in our study. GPUs are, unsurprisingly, the optimal platform when high-end hardware is available. Our goal, however, is not to replace such infrastructure but to enable practical inference under strict resource constraints; TranSQL+ targets precisely these low-memory, heterogeneous environments where a GPU is unavailable or impractical.

Summary. Overall, these results show that TranSQL+ outperforms existing inference frameworks on resource-constrained hardware. Its template-based SQL generation plus relational optimization delivers competitive speed without hardware-specific engineering or external runtimes, showing that databases can serve as practical, portable LLM engines. Unless otherwise noted, we use DuckDB as the backend in the remaining experiments.

5.3 Performance Insights (Q2)

To assess the efficiency gains from our post-optimizations, we (i) provide an operator-level performance breakdown to show where time is saved, (ii) run ablations to isolate each optimization's contribution, and (iii) explore ROW2COL chunking parameters to quantify sensitivity and guide future tuning. All experiments rely solely on the database's native behavior (e.g., vectorized execution by default); we do not enable any special features beyond defaults.

Performance Breakdown by Database Operations. Fig. 9 shows that our post-optimization strategy reduces total execution time by 64.8% compared to unoptimized SQL queries. Optimizations we discussed in Sec. 4 aim to fully utilize parallelization and reduce cardinality of intermediate results, particularly in hash joins and group-by aggregations. The operator-level breakdown confirms the impact of these optimizations: join time is reduced by 62.1%, and group-by operations see a 70.3% reduction. Moreover, our use of optimized CTEs allows the database's vectorized execution engine to exploit greater parallelism. This results in a 66.4% reduction in projection time, highlighting improved cache locality and reduced function call overhead.

Ablation Study for Post-optimizations. We evaluate the individual and combined impact of our three post optimizations (i.e. temp view elimination (denoted as CTE), table fusion, and ROW2COL) in Fig. 10. Each optimization outperforms a direct SQL translation baseline. In prefill, applying all three together yields the greatest speedup for the Llama3 model, whereas the MoE model see smaller gains: longer prompts activate more experts, incurring I/O overhead that these optimizations cannot fully eliminate. During decoding, MoE benefits more because only a few experts are accessed per

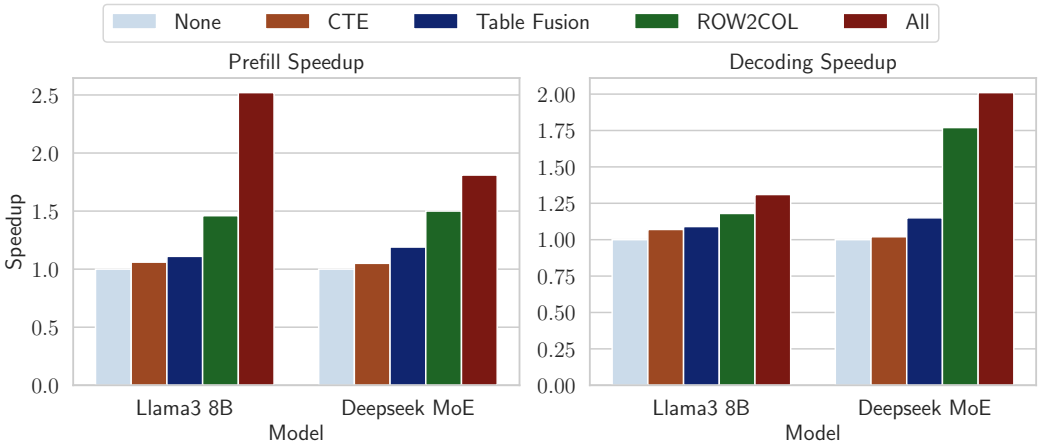


Fig. 10. Ablation study for three post optimizations.

token, allowing our optimizations to effectively exploit parallelism of the vectorized execution engine.

Among the three optimizations, CTE provides only modest improvements, since modern databases already lazily evaluate temporary views; materialized CTEs simply avoid repeated schema creation and view re-evaluation. Table fusion delivers noticeable speedups in attention layers by consolidating Q/K/V weight tables into a single relation, thereby reducing join overhead. ROW2COL has the most dramatic effect in both stages by exploiting columnar storage and SIMD-parallel operations to minimize scan volume. Together, these targeted optimizations leverage the strengths of vectorized, columnar execution engines to automatically accelerate SQL codes generated by TranSQL$^+$.

Effectiveness of ROW2COL. In Sec. 4, we introduced ROW2COL, which pivots chunk indices into columns to cut join cardinality and exploit vectorized execution. This creates a trade-off: too many projections in one query can exceed parallelism, while too many subqueries increase I/O. To study this balance, we benchmark speedups over unoptimized SQL across `#projection` × `#subquery` configurations and CPU core counts. The tested matrices are 25×4096 and $4096 \times 143{,}336$ with chunk size 64 (64 chunks total).

Fig. 11 shows strong sensitivity to configuration. On 4 cores, 16 projections with 4 subqueries yields the best speedup (2.52×), and most settings still outperform the baseline—confirming ROW2COL's effectiveness. However, no single configuration is optimal across all cores, and computing all pivoted columns in one query is consistently suboptimal. We therefore fix 16/4 in our experiments and leave a hardware-aware cost model for future work.

5.4 Influence of Memory Capacity (Q3)

This section analyzes TranSQL$^+$'s memory usage and disk I/O patterns during inference. First, we vary available memory to measure its impact on performance and on the proportion of I/O. Next, we track the total data loaded from disk into memory, revealing the workload imposed by dynamic data paging.

Layer-wise Performance Breakdown. Fig. 12 breaks down runtime by component (normalization, attention, feed-forward) for Llama3 and DeepSeek MoE.

In the **prefill phase**, reducing memory from 16GB to 8GB slows execution, especially in attention and feed-forward layers. The pie charts show the disk-I/O fraction nearly doubling at the lower

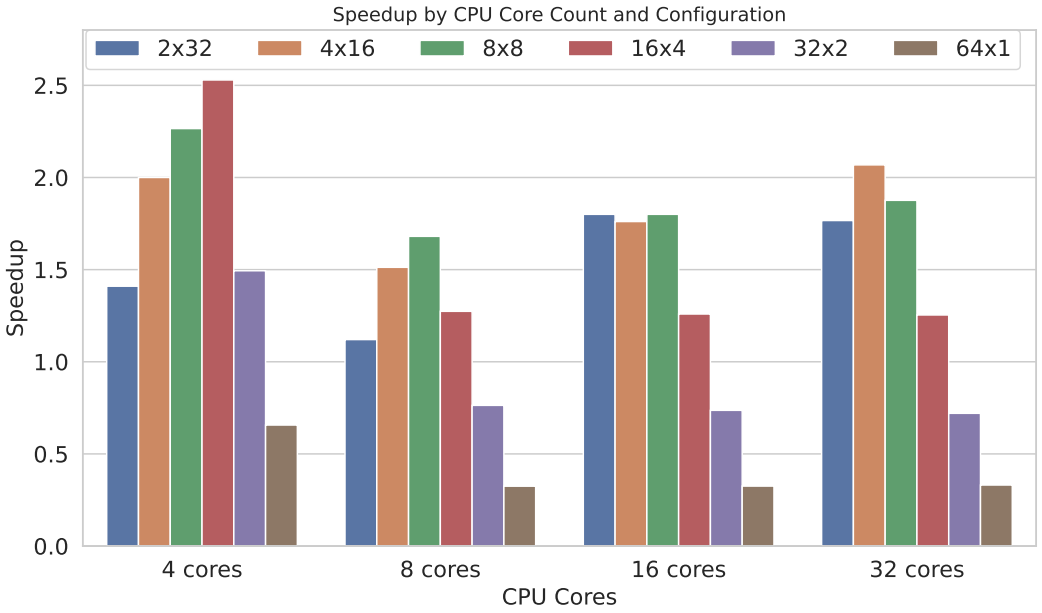


Fig. 11. ROW2COL optimization improves matrix multiplication performance across most configurations. Based on empirical results for 4 CPU cores, we adopt the 16×4 configuration as a balanced default.

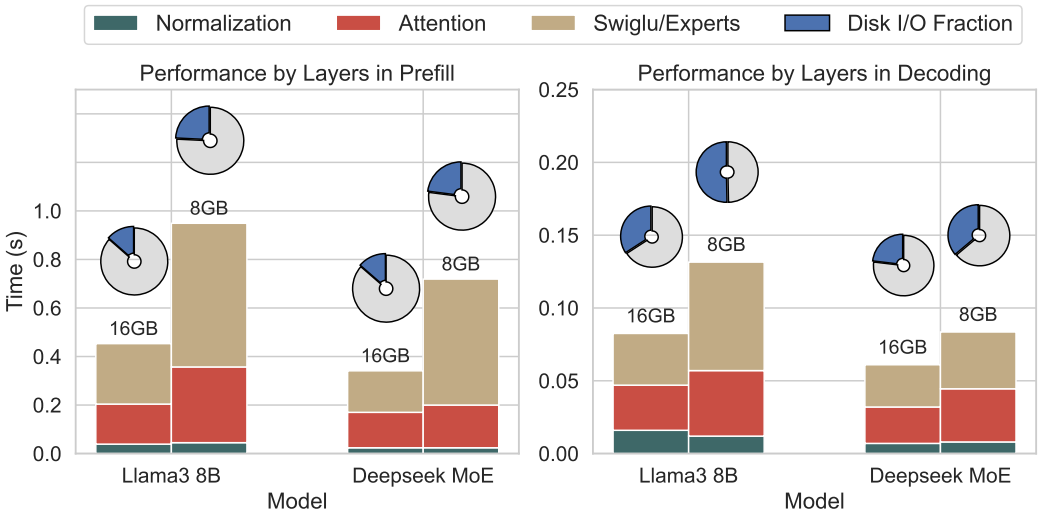


Fig. 12. Layer-wise runtime and disk-I/O fraction for 25-token prompts under varying memory. Prefill performance degrades with less memory (nearly 2× disk I/O). MoE decoding scales better since only a subset of experts is active, reducing I/O.

memory setting. Although I/O is not dominant, each read/write can stall the CPU, so the impact exceeds the fraction alone. DeepSeek MoE exhibits the same pattern, as longer inputs activate more

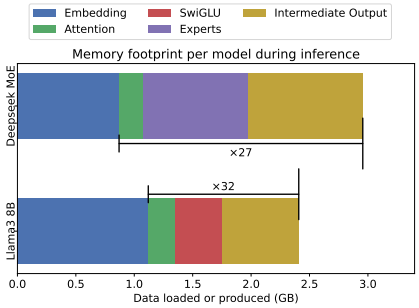


Fig. 13. Prefill memory usage with 50-token input.

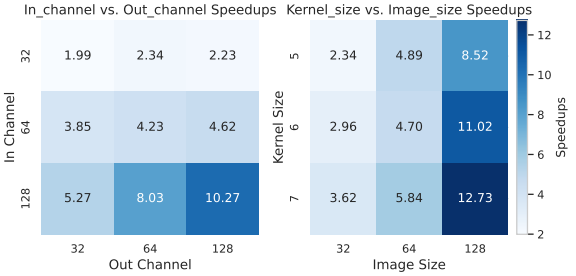


Fig. 14. In the left figure, kernel size is fixed at 5 and image size at 64; in the right, input channels are fixed at 32 and output channels at 64. SQL generated by TranSQL$^+$ consistently outperforms DL2SQL and demonstrates strong scalability.

experts, pushing additional weights through the buffer pool and increasing swap traffic. Sufficient memory helps avoid frequent memory–disk swapping and preserves performance.

In the **decoding phase**, the effect of reduced memory is smaller. Per-token work is modest, so latency is far lower than in prefill; however, the same weights must still be streamed, raising the I/O share, which nearly doubles when memory is halved. Absolute latencies for normalization and attention remain stable; the MoE feed-forward layer is likewise steady because only a small subset of experts is active per token.

Memory Footprint by Layers. Fig. 13 reports the per-layer data loaded during prefill; we omit decoding, whose weight I/O closely mirrors prefill. The embedding table is accessed only for prompt tokens, and an index prevents full scans, so its memory footprint is much smaller than the embedding table size. DeepSeek MoE's expert layer consumes far more memory than Llama3's SwiGLU because multiple experts are active during prefill. Intermediate activations are also substantial. Under limited memory, these weights and activations are repeatedly swapped between the buffer pool and disk, which drives I/O—56.1GB for DeepSeek MoE and 41.2GB for Llama3 8B. Halving memory roughly doubles the swapped volume and nearly doubles prefill latency (Fig. 12). Latency degradation is milder during decoding, since KV vectors are cached and only a single token is processed, making intermediate activations negligible. DuckDB's buffer management and latency hiding outperform the simple memory-mapped approach used by our baselines, yielding faster inference with TranSQL$^+$.

5.5 Extensibility Beyond Transformers: Convolution as a Case Study (Q4)

TranSQL$^+$ is designed to be highly extensible to other types of neural network architectures beyond transformers. To demonstrate this generality, we implement a 2D convolution operator within the same template-based SQL generation framework and compare our performance against **DL2SQL** [31], which also translates deep learning computations into SQL. Unlike TranSQL$^+$, DL2SQL does not support self-attention and represents model parameters in a flattened format, lacking the chunk-based representation and vectorization optimizations used in our approach.

Fig. 14 presents the speedups of TranSQL$^+$ with post-optimization over DL2SQL across various convolution configurations. The left heatmap shows speedup trends with increasing input and output channel sizes, while the right illustrates performance changes as image size and kernel size increase. The results clearly indicate that TranSQL$^+$ scales more effectively with computational complexity. As both channel dimensions and kernel/image sizes grow, the performance gap widens

significantly. For example, when convolving large feature maps (e.g., 128×128 images with 7×7 kernels), TranSQL+ achieves a speedup of up to **12.73**× over DL2SQL. This advantage gains from better table design, reduced join cardinality, and improved parallelism through vectorized execution.

Results show that our template-based SQL and post-optimizations extend beyond transformers to convolutions, making TranSQL+ viable for a broader range of deep learning inference.

6 Related Work

Deep learning has traditionally used specialized hardware and frameworks, but recent work embeds neural computation in relational databases, leveraging database optimizations to express inference in SQL. Here we review in-database inference, memory offloading for large models, and relational approaches to linear algebra, laying the groundwork for our extension to LLMs.

6.1 In-Database Neural Network Inference

Early systems [3, 4, 17] integrated neural network inference into databases by implementing operations as user-defined functions (UDFs) that offload computation to external ML frameworks. While this decoupled approach avoids reimplementing neural operators in SQL, it incurs substantial cross-system overhead and limits joint optimization of ML inference and SQL processing.

Recent work explores *relation-centric inference*, mapping neural operations to relational query primitives. DuckBrain [39] and [22] demonstrate SQL-based matrix multiplication for training and inference, suggesting that databases can host neural computation. [22] targets efficient basic linear algebra on distributed databases with cost estimation and a specialized planner, whereas [39] exploits a single system's intrinsic capabilities for both training and inference. Neither supports complex non-linear kernels such as self-attention or residual connections, so we do not compare them directly. Closer to our scope, **DL2SQL** [31] expresses each layer as a SQL query over tensor relations, extending beyond basic matrix operations to non-linear kernels (e.g., convolutions, residual connections). Because DL2SQL lacks self-attention, we implement a convolution kernel in TranSQL+ and compare against DL2SQL's convolution (Sec. 5.5). Collectively, these systems underscore a shift toward executing inference inside the database rather than in separate ML frameworks.

Our work builds on this research by extending the relation-centric paradigm to large language models (LLMs). We translate the full suite of LLM operations—including transformer attention and feed-forward layers—into SQL. This approach is, to our knowledge, the first to use a general-purpose SQL engine as the inference engine for LLMs, as earlier in-database systems have focused on smaller models or did not target today's foundation-scale models.

6.2 Offloading and Paging Large Models to RAM and Storage

Running LLMs on commodity or resource-constrained hardware has motivated techniques for memory offloading and paging of model weights. **FlexGen** [43] enables large models (e.g., 175-billion-parameter GPT-style networks) to run on a single GPU by partitioning data across GPU DRAM, CPU RAM, and disk. **LLMFlash** [9] leverages high-capacity NVMe flash storage to extend main memory, using techniques like windowing and row–column bundling to reduce data transfers and improve speed. **FlexInfer** [14] focuses on on-device inference by dynamically offloading model layers between GPU and CPU, achieving significant throughput improvements on limited-memory devices. All these techniques highlight the benefits of efficient memory management for large models. Our approach adopts a similar strategy but within a database system. Since relational databases have built-in paging mechanisms (e.g., buffer pools and caches), representing LLM weights as relational tables allows the database to transparently manage memory, reducing engineering effort and loosening the coupling with specific hardware architectures.

6.3 Linear Algebra in Relational Systems

Our work is also related to research on performing matrix computations with relational query processors. Prior work has leveraged relational databases to execute operations like matrix multiplication [22, 23, 34] and other linear algebra kernels [10, 30] by treating matrices as relational data. Luo *et al.* [32] showed that with minor extensions, a database can serve as a high-performance linear algebra engine, matching the scalability of specialized systems. However, these methods typically focus on blocked-matrix multiplication and struggle with non-linear computations (e.g., softmax, layer normalization) required by LLMs.

In contrast, approaches like tensor relational algebra [24, 33] map tensor operations to relational algebra, enabling some iterative ML tasks to be expressed as SQL queries. Yet, such systems often necessitate substantial changes to underlying compilers and optimizers, limiting their compatibility with the current deep learning ecosystem. Our method, TranSQL$^+$, generates SQL code from an abstract operator level, striking a balance between extensibility, ease of integration, and leveraging existing database engines.

7 Conclusion and Discussion

In this work, we present TranSQL$^+$, a template-based framework that compiles LLM computation graphs into executable SQL for end-to-end inference entirely inside a relational database—the first, to our knowledge, to do so without external ML libraries, GPUs, or custom engines. Our results show that principled code generation and optimization enable modern databases to serve as practical engines for resource-constrained LLM inference. Built purely on standard SQL, TranSQL$^+$ is lightweight and portable, easily deployable on laptops, smartphones, and edge hardware, and it integrates smoothly with embedded databases to provide scalable, low-cost inference without bespoke engineering. While not intended to match GPU frameworks in speed, TranSQL$^+$ targets a different trade-off: accessible, efficient inference on CPU-only platforms where traditional solutions fall short.

Although prefill latency remains high—calling for further DB-side optimizations—the fast decoding stage already gives TranSQL$^+$ practical value. Recent work offloads only decoding [52] or runs a small edge model collaboratively with a large cloud model [18]. Our approach offers an alternative runtime for the edge component: fully portable, SQL-based, and immediately useful in real deployments. Future work includes enabling quantized arithmetic in databases to further reduce the storage overhead, memory usage, and improve performance. More broadly, relational representations of LLMs may open doors for analytical inspection—treating models as data to better understand their behavior.

Acknowledgment

RH acknowledges the support received through the Dutch Research Council (VI.Veni.222.439). This work has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe Research and Innovation programme under Grant No. 101192750. This work is partially supported by NSF award No. 2112631.

References

[1] 2017. Open Neural Network Exchange: The open standard for machine learning interoperability. https://github.com/onnx/onnx.
[2] 2023. Most Widely Deployed and Used Database Engine. https://sqlite.org/mostdeployed.html.
[3] 2024. PostgresML. postgresml.org.
[4] 2024. SQL machine learning documentation. https://learn.microsoft.com/en-us/sql/machine-learning/?view=sql-server-ver16.

[5] 2025. ARM v8 Manual. https://developer.arm.com/documentation/ddi0553/latest/.

[6] 2025. llama.cpp. https://github.com/ggml-org/llama.cpp.

[7] 2025. Risc-V Instruction Set. https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications#ISA-Specifications.

[8] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 4895–4901. doi:10.18653/V1/2023.EMNLP-MAIN.298

[9] Keivan Alizadeh, Seyed-Iman Mirzadeh, Dmitry Belenko, S. Khatamifard, Minsik Cho, Carlo C. del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 12562–12584. doi:10.18653/V1/2024.ACL-LONG.678

[10] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards linear algebra over normalized data. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1214–1225. doi:10.14778/3137628.3137633

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, and et al. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[12] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 1280–1297. doi:10.18653/V1/2024.ACL-LONG.70

[13] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2025. AgentDojo: a dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2636, 26 pages.

[14] Hongchao Du, Shangyu Wu, Arina Kharlamova, Nan Guan, and Chun Jason Xue. 2025. FlexInfer: Breaking Memory Constraint via Flexible and Efficient Offloading for On-Device LLM Inference. *CoRR* abs/2503.03777 (2025). doi:10.48550/ARXIV.2503.03777 arXiv:2503.03777

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

[16] David Eigen, Marc'Aurelio Ranzato, and Ilya Sutskever. 2013. Learning factored representations in a deep mixture of experts. *arXiv preprint arXiv:1312.4314* (2013).

[17] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 755–768. doi:10.1145/3318464.3386137

[18] Zixu Hao, Huiqiang Jiang, Shiqi Jiang, Ju Ren, and Ting Cao. 2024. Hybrid SLM and LLM for Edge-Cloud Collaborative Inference. In *Proceedings of the Workshop on Edge and Mobile Foundation Models* (Minato-ku, Tokyo, Japan) *(EdgeFM '24)*. Association for Computing Machinery, New York, NY, USA, 36–41. doi:10.1145/3662006.3662067

[19] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[20] Sándor Héman, Marcin Zukowski, Arjen P. de Vries, and Peter A. Boncz. 2007. Efficient and Flexible Information Retrieval using MonetDB/X100. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 96–101. http://cidrdb.org/cidr2007/papers/cidr07p10.pdf

[21] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. 2021. PyTorch. *Programming with TensorFlow: solution for edge computing applications* (2021), 87–104.

[22] 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.

[23] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS. *Proc. VLDB Endow.* 12, 7 (2019), 822–835. doi:10.14778/3317315.3317323

[24] Chris Jermaine. 2021. The Tensor-Relational Algebra, and Other Ideas in Machine Learning System Design. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management* (Tampa, FL, USA) *(SSDBM '21)*. Association for Computing Machinery, New York, NY, USA, 270. doi:10.1145/3468791.3472262

[25] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[26] Songtao Jiang, Tuo Zheng, Yan Zhang, Yeying Jin, Li Yuan, and Zuozhu Liu. 2024. Med-MoE: Mixture of Domain-Specific Experts for Lightweight Medical Vision-Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 3843–3860. doi:10.18653/v1/2024.findings-emnlp.221

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[29] Haitao Li, Qingyao Ai, Jia Chen, Qian Dong, Zhijing Wu, and Yiqun Liu. 2025. BLADE: Enhancing Black-Box Large Language Models with Small Domain-Specific Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 23 (Apr. 2025), 24422–24430. doi:10.1609/aaai.v39i23.34620

[30] Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1571–1588. doi:10.1145/3299869.3319878

[31] 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.

[32] Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, and Christopher Jermaine. 2018. Scalable Linear Algebra on a Relational Database System. *SIGMOD Rec.* 47, 1 (Sept. 2018), 24–31. doi:10.1145/3277006.3277013

[33] Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, and Christopher Jermaine. 2018. Scalable Linear Algebra on a Relational Database System. *SIGMOD Rec.* 47, 1 (Sept. 2018), 24–31. doi:10.1145/3277006.3277013

[34] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *SIGMOD '21*. ACM, 1222–1234.

[35] 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*.

[36] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. doi:10.1145/3299869.3320212

[37] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[38] Maximilian E Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL-Compiling a Declarative Machine Learning Language to SQL and Python.. In *EDBT*. 562–565.

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

[40] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744. doi:10.14778/3685800.3685802

[41] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. arXiv:1911.02150 [cs.NE] https://arxiv.org/abs/1911.02150

[42] Noam Shazeer. 2020. GLU Variants Improve Transformer. *CoRR* abs/2002.05202 (2020). arXiv:2002.05202 https://arxiv.org/abs/2002.05202

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). doi:10.48550/ARXIV.2302.13971 arXiv:2302.13971

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[46] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. *CoRR* abs/2406.06282 (2024). doi:10.48550/ARXIV.2406.06282 arXiv:2406.06282

[47] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[48] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, , and et al. 2024. Qwen2 Technical Report. *CoRR* abs/2407.10671 (2024). doi:10.48550/ARXIV.2407.10671 arXiv:2407.10671

[49] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 10471–10506. doi:10.18653/v1/2024.findings-acl.624

[50] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL] https://arxiv.org/abs/2309.11998

[51] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

[52] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

[53] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.

Received April 2025; revised July 2025; accepted August 2025