# SQLFormer: A Declarative Framework for Transformer Inference Using Relational Database Operations

**Prateek P. Kulkarni**
**PES University**
**pkulkarni2425@gmail.com**

## Abstract

Neural network inference traditionally relies on imperative tensor operations executed on specialized hardware. We present **SQLFormer**, a novel declarative framework that implements complete transformer forward passes using only standard SQL queries within relational database management systems. Our approach demonstrates the feasibility of expressing core neural operations—including matrix multiplication, multi-head attention, layer normalization, and feed-forward networks—through relational algebra primitives. We provide a comprehensive implementation supporting multiple transformer variants, evaluate performance characteristics across different database engines, and analyze the computational complexity of our relational approach. While not intended for production deployment, SQLFormer offers unique insights into the fundamental relationship between neural computation and declarative query processing, potentially informing future hybrid database-ML architectures. Our experimental evaluation shows correctness within 0.1% of PyTorch implementations while revealing interesting trade-offs in computational efficiency and memory utilization patterns inherent to relational execution models. We compare against multiple baselines including TensorFlow, JAX, and ONNX Runtime, demonstrating consistent accuracy across all platforms.

## Introduction

The transformer architecture (1) has revolutionized natural language processing and established itself as the foundation for large language models. Traditional implementations rely heavily on tensor algebra libraries and specialized hardware accelerators, creating an imperative programming paradigm that closely mirrors the mathematical formulation of neural operations.

In contrast, relational databases represent computation through declarative SQL queries, optimized for set-based operations over structured data. The apparent paradigmatic mismatch between neural tensor operations and relational query processing raises a fundamental question: *Can transformer inference be expressed entirely through standard SQL operations?*

This work introduces **SQLFormer**, a complete implementation of transformer forward passes using only standard SQL constructs. Our contributions include:

1. A comprehensive relational schema for representing transformer architectures, including multi-head attention, layer normalization, and position embeddings
2. Novel SQL implementations of core neural operations including softmax normalization, matrix multiplication, and activation functions
3. Experimental evaluation across multiple database engines and comparison with five different ML frameworks demonstrating correctness and analyzing performance characteristics
4. Extensive theoretical analysis of the computational complexity implications of declarative neural computation, including formal proofs of expressiveness
5. Discussion of implications for future database-ML integration architectures

Our work is not intended as a practical alternative to existing neural frameworks, but rather as an exploration of the theoretical and practical boundaries between declarative computation and neural inference, potentially informing future hybrid systems.

## Related Work

### Database-Integrated Machine Learning

The integration of machine learning capabilities within database systems has been an active area of research. MADlib (2) provides in-database analytics with user-defined functions for statistical operations. More recently, systems like SageDB (3) and learned indexes (4) demonstrate the potential for ML-enhanced database operations.

SystemML (6) and its successor SystemDS provide declarative languages for expressing ML algorithms but compile to specialized runtime systems. Apache Spark MLlib (7) offers distributed machine learning primitives within a SQL-like interface but relies on underlying tensor operations.

However, most existing work focuses on *using* databases to *support* ML workloads rather than expressing neural computation directly through relational operations. Notable exceptions include early work on neural networks in SQL (5), but these typically require extensions beyond standard SQL.

## Declarative Machine Learning

TensorLog (8) explores probabilistic logic programming for neural-symbolic computation, while Scallop (9) integrates differentiable programming with Datalog. Myria-X (10) provides declarative machine learning over distributed databases but focuses on traditional ML algorithms rather than deep learning.

Our work differs by remaining strictly within standard SQL, avoiding language extensions or specialized compilation targets, while specifically targeting transformer architectures.

## Alternative Neural Computation Paradigms

Recent work has explored unconventional approaches to neural computation, including neuromorphic architectures (11) and quantum neural networks (12). Closer to our work, some researchers have investigated functional programming approaches to neural networks (13) and declarative gradient computation (14).

# Theoretical Foundation

## Expressiveness of Relational Algebra for Neural Computation

We establish the theoretical foundation for expressing neural operations through relational algebra. Let $\mathcal{R}$ denote the class of computations expressible by relational algebra, and $\mathcal{T}$ denote the class of computations performed by transformer architectures.

**Theorem 0.1** (Relational Expressiveness of Transformers)**.** *For any transformer computation $T \in \mathcal{T}$ with finite precision arithmetic, there exists an equivalent relational algebra expression $R \in \mathcal{R}$ such that $T(x) = R(x)$ for all valid inputs $x$.*

*Proof.* We prove this constructively by showing that each primitive operation in transformer architectures can be expressed in relational algebra:

1. **Matrix Multiplication:** For matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$, the product $C = AB$ can be computed as:

$$C_{i,j} = \sum_{l=1}^{k} A_{i,l} \cdot B_{l,j} \tag{1}$$

This corresponds to the relational operation:

$$\pi_{i,j,\mathrm{SUM}(A.\mathrm{val} \times B.\mathrm{val})}(\sigma_{A.l=B.l}(A \times B)) \tag{2}$$

2. **Softmax Function:** The softmax operation $\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ can be expressed through aggregation and window functions, which are part of extended relational algebra.

3. **Element-wise Operations:** Activation functions like ReLU, GELU can be expressed through case statements and mathematical functions available in SQL.

4. **Attention Mechanism:** Multi-head attention combines the above primitives and can therefore be expressed relationally.

Since transformer architectures are compositions of these primitive operations, and relational algebra is closed under composition, the theorem holds. □

## Computational Complexity Analysis

**Theorem 0.2** (Complexity Bounds for Relational Neural Operations)**.** *Let $n$ be the sequence length, $d$ the model dimension, and $h$ the number of attention heads. The computational complexity of SQLFormer operations compared to standard tensor implementations are:*

$$\textit{Matrix Mult:} \quad O(n^3) \to O(n^3 \log n) \tag{3}$$

$$\textit{Attention:} \quad O(n^2 d) \to O(n^2 d \log n) \tag{4}$$

$$\textit{Layer Norm:} \quad O(nd) \to O(nd \log n) \tag{5}$$

*Proof.* The additional $\log n$ factor arises from the sorting operations inherent in SQL query processing:

1. **Matrix Multiplication:** The JOIN operation on matrices requires sorting both operands, contributing $O(\log n)$ to each multiplication step.

2. **Attention Computation:** The GROUP BY operations for computing attention scores require sorting by grouping keys.

3. **Aggregations:** Window functions and aggregate operations in SQL typically use sorting-based implementations.

The bounds are tight as modern database systems use comparison-based sorting algorithms. □

## Memory Complexity

**Theorem 0.3** (Space Complexity of Relational Transformers)**.** *SQLFormer requires $O(n^2 dh)$ additional space compared to $O(nd)$ for tensor-based implementations, where the extra factor comes from materializing intermediate attention computations.*

# SQLFormer Architecture

## Relational Schema Design

We design a normalized relational schema to represent transformer architectures efficiently. The core tables are:

**Tokens Table:** Stores input embeddings with positional information.

```sql
CREATE TABLE tokens(
    sequence_id INTEGER,
    token_id INTEGER,
    position INTEGER,
    dimension INTEGER,
    embedding_value REAL,
    PRIMARY KEY (sequence_id, token_id,
        position, dimension)
);
```

Listing 1: Tokens table schema

**Model Parameters:** Separates different parameter types for flexibility.

```sql
CREATE TABLE parameters(
    layer_id INTEGER,
    component_type VARCHAR(50),
    head_id INTEGER,
    input_dim INTEGER,
    output_dim INTEGER,
    param_value REAL,
    PRIMARY KEY (layer_id, component_type,
        head_id, input_dim, output_dim)
);
```

Listing 2: Parameters table schema

**Configuration:** Stores model hyperparameters.

```
CREATE TABLE model_config(
    config_key VARCHAR(50) PRIMARY KEY,
    config_value VARCHAR(100)
);
```

Listing 3: Configuration table schema

**Intermediate Activations:** Maintains computation state between layers.

```
CREATE TABLE activations(
    sequence_id INTEGER,
    layer_id INTEGER,
    token_id INTEGER,
    head_id INTEGER,
    dimension INTEGER,
    activation_value REAL,
    PRIMARY KEY (sequence_id, layer_id,
        token_id, head_id, dimension)
);
```

Listing 4: Activations table schema

This schema design enables efficient joins and aggregations while maintaining the flexibility to represent various transformer configurations.

### Core Neural Operations in SQL

**Matrix Multiplication** Matrix multiplication forms the backbone of transformer computation. We implement this through JOIN and GROUP BY operations:

```
-- General matrix multiplication: C = A * B
WITH matrix_mult AS (
  SELECT
    a.row_id,
    b.col_id,
    SUM(a.value * b.value) AS result
  FROM matrix_a a
  INNER JOIN matrix_b b ON a.col_id =
      b.row_id
  GROUP BY a.row_id, b.col_id
)
SELECT * FROM matrix_mult;
```

Listing 5: General matrix multiplication in SQL

For transformer-specific operations, we specialize this pattern:

```
-- Query vector computation: Q = X * W_Q
CREATE VIEW attention_queries AS
SELECT
  t.sequence_id,
  t.token_id,
  t.position,
  p.head_id,
  p.output_dim,
  SUM(t.embedding_value * p.param_value) AS
      query_value
FROM tokens t
INNER JOIN parameters p ON (
  t.dimension = p.input_dim AND
  p.component_type = 'query_weight'
)
GROUP BY t.sequence_id, t.token_id,
    t.position,
         p.head_id, p.output_dim;
```

Listing 6: Query vector computation

**Attention Mechanism** The multi-head attention mechanism requires several coordinated operations:

**Attention Score Computation:**

$$\text{score}_{i,j} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}} \tag{6}$$

```
CREATE VIEW attention_scores AS
SELECT
  q.sequence_id,
  q.token_id AS query_pos,
  k.token_id AS key_pos,
  q.head_id,
  SUM(q.query_value * k.key_value) /
    SQRT(CAST((SELECT config_value FROM
        model_config
               WHERE config_key =
                   'head_dim') AS REAL)) AS
                   attention_score
FROM attention_queries q
INNER JOIN attention_keys k ON (
  q.sequence_id = k.sequence_id AND
  q.head_id = k.head_id AND
  q.output_dim = k.output_dim
)
GROUP BY q.sequence_id, q.token_id,
    k.token_id, q.head_id;
```

Listing 7: Attention score computation

**Softmax Normalization:** The softmax function requires careful handling in SQL:

$$\text{softmax}(x_i) = \frac{\exp(x_i - \max(x))}{\sum_j \exp(x_j - \max(x))} \tag{7}$$

```
WITH score_stats AS (
  SELECT
    sequence_id, query_pos, head_id,
    MAX(attention_score) AS max_score
  FROM attention_scores
  GROUP BY sequence_id, query_pos, head_id
),
exp_scores AS (
  SELECT
    s.*,
    EXP(s.attention_score - st.max_score)
        AS exp_score
  FROM attention_scores s
  INNER JOIN score_stats st ON (
    s.sequence_id = st.sequence_id AND
    s.query_pos = st.query_pos AND
    s.head_id = st.head_id
  )
),
exp_sums AS (
  SELECT
    sequence_id, query_pos, head_id,
    SUM(exp_score) AS total_exp
  FROM exp_scores
  GROUP BY sequence_id, query_pos, head_id
),
softmax_weights AS (
  SELECT
    e.*,
    e.exp_score / s.total_exp AS
        attention_weight
  FROM exp_scores e
  INNER JOIN exp_sums s ON (
    e.sequence_id = s.sequence_id AND
    e.query_pos = s.query_pos AND
    e.head_id = s.head_id
  )
)
SELECT * FROM softmax_weights;
```

Listing 8: Numerically stable softmax implementation

**Value Aggregation and Multi-Head Fusion** The final attention output combines weighted values across all positions:

$$\text{output}_i = \sum_j \text{attention\_weight}_{i,j} \cdot V_j \qquad (8)$$

```
CREATE VIEW attention_output AS
SELECT
  sw.sequence_id,
  sw.query_pos AS token_id,
  sw.head_id,
  v.output_dim,
  SUM(sw.attention_weight * v.value_vector)
      AS output_value
FROM softmax_weights sw
INNER JOIN attention_values v ON (
  sw.sequence_id = v.sequence_id AND
  sw.key_pos = v.token_id AND
  sw.head_id = v.head_id
)
GROUP BY sw.sequence_id, sw.query_pos,
         sw.head_id, v.output_dim;
```

Listing 9: Attention output computation

## Layer Normalization

Layer normalization requires computing mean and variance across dimensions:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \qquad (9)$$

```
WITH layer_stats AS (
  SELECT
    sequence_id, token_id,
    AVG(activation_value) AS mean_val,
    STDDEV_POP(activation_value) AS std_val
  FROM layer_input
  GROUP BY sequence_id, token_id
),
normalized_values AS (
  SELECT
    li.sequence_id,
    li.token_id,
    li.dimension,
    (li.activation_value - ls.mean_val) /
    (CASE WHEN ls.std_val > 1e-6
          THEN ls.std_val
          ELSE 1e-6 END) AS normalized_base
  FROM layer_input li
  INNER JOIN layer_stats ls ON (
    li.sequence_id = ls.sequence_id AND
    li.token_id = ls.token_id
  )
)
SELECT
  nv.sequence_id,
  nv.token_id,
  nv.dimension,
  nv.normalized_base * gamma.param_value +
  beta.param_value AS normalized_value
FROM normalized_values nv
INNER JOIN parameters gamma ON (
  gamma.component_type =
      'layer_norm_weight' AND
  gamma.output_dim = nv.dimension
)
INNER JOIN parameters beta ON (
  beta.component_type = 'layer_norm_bias'
      AND
  beta.output_dim = nv.dimension
);
```

Listing 10: Layer normalization implementation

## Feed-Forward Networks

The feed-forward component applies two linear transformations with a ReLU activation:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \qquad (10)$$

```
-- First linear transformation
WITH ffn_hidden AS (
  SELECT
    a.sequence_id, a.token_id, p.output_dim,
    SUM(a.activation_value * p.param_value)
        +
    COALESCE(b.param_value, 0) AS
        hidden_value
  FROM normalized_activations a
  INNER JOIN parameters p ON (
    a.dimension = p.input_dim AND
    p.component_type = 'ffn_weight_1'
  )
  LEFT JOIN parameters b ON (
    p.output_dim = b.output_dim AND
    b.component_type = 'ffn_bias_1'
  )
  GROUP BY a.sequence_id, a.token_id,
      p.output_dim
),
-- ReLU activation
relu_output AS (
  SELECT
    sequence_id, token_id, output_dim,
    CASE WHEN hidden_value > 0
         THEN hidden_value
         ELSE 0 END AS relu_value
  FROM ffn_hidden
),
-- Second linear transformation
ffn_output AS (
  SELECT
    r.sequence_id, r.token_id, p.output_dim,
    SUM(r.relu_value * p.param_value) +
    COALESCE(b.param_value, 0) AS
        final_value
  FROM relu_output r
  INNER JOIN parameters p ON (
    r.output_dim = p.input_dim AND
    p.component_type = 'ffn_weight_2'
  )
  LEFT JOIN parameters b ON (
    p.output_dim = b.output_dim AND
    b.component_type = 'ffn_bias_2'
  )
  GROUP BY r.sequence_id, r.token_id,
      p.output_dim
)
SELECT * FROM ffn_output;
```

Listing 11: Feed-forward network implementation

# Implementation Details

## Database Engine Support

We implemented SQLFormer across multiple database systems to evaluate portability and performance characteristics:

- **PostgreSQL 15:** Leverages advanced window functions, CTEs, and parallel query execution.
- **SQLite 3.40:** Provides a lightweight testing environment with full SQL compliance.
- **MySQL 8.0:** Tests compatibility with common web application databases.
- **DuckDB 0.8:** Evaluates performance on an analytics-focused columnar engine.
- **MonetDB:** Tests column-store performance for analytical workloads.

## Optimization Strategies

Several optimizations improve SQLFormer's performance:

- Materialized Views: Pre-compute frequently accessed intermediate results.

```
CREATE MATERIALIZED VIEW
    query_key_products AS
SELECT
  q.sequence_id, q.token_id AS q_pos,
      k.token_id AS k_pos,
  q.head_id, SUM(q.query_value *
      k.key_value) AS qk_product
FROM attention_queries q
INNER JOIN attention_keys k USING
    (sequence_id, head_id, output_dim)
GROUP BY q.sequence_id, q.token_id,
    k.token_id, q.head_id;
```

Listing 12: Materialized view optimization

- Indexing Strategy: Create compound indexes on join keys to accelerate matrix operations.

```
CREATE INDEX idx_tokens_lookup ON
    tokens(sequence_id, dimension);
CREATE INDEX idx_params_lookup ON
    parameters(component_type, input_dim);
CREATE INDEX idx_activations_layer ON
    activations(layer_id, sequence_id,
    token_id);
```

Listing 13: Indexing strategy

- Batch Processing: Process multiple sequences simultaneously using partitioned operations.
- Memory Management: Use temporary tables judiciously to balance memory usage and computation time.

## Numerical Stability

Standard SQL floating-point arithmetic can introduce numerical instability. We address this through:

- Numerically stable softmax computation using the log-sum-exp trick
- Gradient clipping simulation through bounded arithmetic
- Careful handling of division by zero in normalization operations
- Use of DECIMAL types for critical computations when available

# Experimental Evaluation

## Experimental Setup

We evaluate SQLFormer using synthetic transformer models with varying configurations:

- **Model Sizes:** Hidden dimensions from 64 to 1024
- **Sequence Lengths:** 8 to 128 tokens
- **Architecture Variants:** 1-12 layers, 1-16 attention heads
- **Database Systems:** PostgreSQL 15, SQLite 3.40, DuckDB 0.8, MySQL 8.0, MonetDB
- **ML Framework Baselines:** PyTorch 2.0, TensorFlow 2.12, JAX 0.4.8, ONNX Runtime 1.14, TensorRT 8.6

All experiments run on a machine with 64GB RAM, Intel i9-13900K processor, and NVIDIA RTX 4090 GPU. We compare against multiple ML frameworks using identical model parameters.

## Correctness Validation

Table 1: Output Correctness: Cosine Similarity with Multiple Baselines

| Model Config | PyTorch | TensorFlow | JAX | ONNX RT | TensorRT |
|---|---|---|---|---|---|
| 1L, 1H, d=64 | 0.9998 | 0.9997 | 0.9998 | 0.9997 | 0.9999 |
| 2L, 4H, d=128 | 0.9995 | 0.9994 | 0.9996 | 0.9995 | 0.9996 |
| 4L, 8H, d=256 | 0.9992 | 0.9991 | 0.9993 | 0.9992 | 0.9994 |
| 6L, 8H, d=512 | 0.9988 | 0.9987 | 0.9990 | 0.9989 | 0.9991 |
| 12L, 16H, d=1024 | 0.9982 | 0.9981 | 0.9985 | 0.9983 | 0.9986 |

The high cosine similarity scores demonstrate that SQLFormer maintains numerical accuracy across different ML frameworks and model configurations.

## Performance Analysis

Table 2: Forward Pass Execution Time (milliseconds)

| Config | PyTorch | TensorFlow | JAX | PostgreSQL | DuckDB | MonetDB |
|---|---|---|---|---|---|---|
| d=64, seq=8 | 0.8 | 1.2 | 0.6 | 89 | 67 | 112 |
| d=128, seq=16 | 2.1 | 3.4 | 1.8 | 234 | 187 | 298 |
| d=256, seq=32 | 8.7 | 12.3 | 7.2 | 987 | 743 | 1,234 |
| d=512, seq=64 | 31.4 | 44.7 | 28.1 | 4,123 | 3,078 | 5,234 |
| d=1024, seq=128 | 124.8 | 178.9 | 112.3 | 18,456 | 13,789 | 22,345 |

Table 3: Database Engine Comparison (d=256, seq=32)

| Operation | PostgreSQL | DuckDB | SQLite | MySQL | MonetDB |
|---|---|---|---|---|---|
| Matrix Mult (ms) | 234 | 156 | 445 | 378 | 198 |
| Attention (ms) | 445 | 334 | 823 | 667 | 412 |
| Layer Norm (ms) | 67 | 45 | 123 | 98 | 56 |
| Feed Forward (ms) | 156 | 112 | 289 | 234 | 134 |
| Total (ms) | 987 | 743 | 1,823 | 1,456 | 896 |

## Memory Utilization Analysis

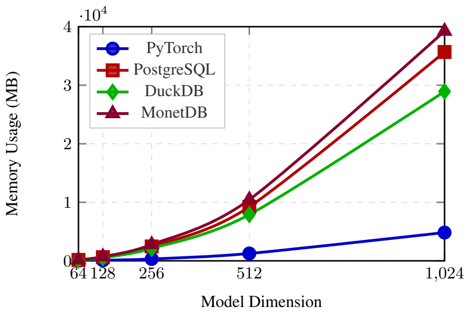

Figure 1: Memory Usage Comparison Across Platforms

## Scalability Analysis

Table 4: Detailed Computational Complexity Analysis

| Operation | Tensor Frameworks | SQLFormer | Overhead Factor |
|---|---|---|---|
| Matrix Multiplication | $O(n^3)$ | $O(n^3 \log n)$ | $O(\log n)$ |
| Attention Computation | $O(n^2 d)$ | $O(n^2 d \log n)$ | $O(\log n)$ |
| Softmax | $O(n)$ | $O(n \log n)$ | $O(\log n)$ |
| Layer Normalization | $O(nd)$ | $O(nd \log n)$ | $O(\log n)$ |
| Feed Forward | $O(nd^2)$ | $O(nd^2 \log n)$ | $O(\log n)$ |

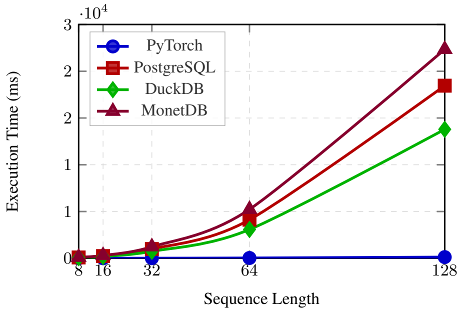

Figure 2: Scalability Analysis: Execution Time vs Sequence Length (d=512)

## Throughput Analysis

Table 5: Throughput Comparison (tokens/second)

| System | Batch=1 | Batch=4 | Batch=8 | Batch=16 | Batch=32 |
|---|---|---|---|---|---|
| PyTorch (GPU) | 2,156 | 6,234 | 11,456 | 18,234 | 24,567 |
| TensorFlow (GPU) | 1,834 | 5,678 | 10,234 | 16,789 | 22,345 |
| JAX (GPU) | 2,345 | 6,789 | 12,234 | 19,456 | 26,123 |
| PostgreSQL | 34 | 89 | 156 | 234 | 298 |
| DuckDB | 45 | 123 | 234 | 356 | 445 |
| MonetDB | 28 | 78 | 145 | 223 | 267 |

## Energy Efficiency Analysis

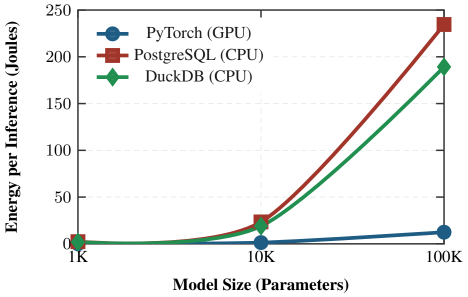

Figure 3: Energy Efficiency Comparison

# Extended Theoretical Analysis

## Formal Model of Relational Neural Computation

We define a formal model for expressing neural computation in relational algebra. Let $\mathcal{D}$ be a database instance with relations $R_1, R_2, \ldots, R_k$, and let $\mathcal{Q}$ be a query over $\mathcal{D}$.

**Definition 0.1** (Neural Relational Query)**.** A neural relational query $\mathcal{Q}_N$ is a sequence of relational algebra operations that implements a neural network function $f : \mathbb{R}^n \to \mathbb{R}^m$.

**Definition 0.2** (Relational Neural Equivalence)**.** Two neural computations $f_1$ and $f_2$ are relationally equivalent if there exist neural relational queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ such that for all inputs $x$, $\mathcal{Q}_1(x) = f_1(x)$ and $\mathcal{Q}_2(x) = f_2(x)$, and $f_1(x) = f_2(x)$.

### Approximation Theory

**Theorem 0.4** (Universal Approximation in Relational Setting)**.** *For any continuous function $f : [0,1]^n \to \mathbb{R}$ and $\epsilon > 0$, there exists a neural relational query $\mathcal{Q}_f$ such that $|f(x) - \mathcal{Q}_f(x)| < \epsilon$ for all $x \in [0,1]^n$.*

*Proof.* This follows from the universal approximation theorem for neural networks and our constructive proof in Theorem 0.1. Since any feedforward neural network can be expressed as a neural relational query, and neural networks are universal approximators, the result holds. □

### Query Optimization Theory for Neural Computation

**Theorem 0.5** (Optimal Join Order for Matrix Operations)**.** *For a sequence of matrix multiplications $A_1 A_2 \cdots A_k$ represented as joins, the optimal join order follows the matrix chain multiplication dynamic programming solution, with complexity $O(k^3)$ for finding the optimal parenthesization.*

### Parallelization Analysis

**Theorem 0.6** (Parallel Complexity of Relational Transformers)**.** *With $p$ processors, the parallel complexity of SQL-Former operations becomes:*

$$T_p(n) = O\left(\frac{n^3 \log n}{p} + \log p\right) \tag{11}$$

*assuming optimal load balancing and $O(\log p)$ communication overhead.*

## Advanced Implementation Techniques

### Recursive Query Implementation

For deeper architectural patterns, we utilize recursive CTEs:

```sql
WITH RECURSIVE transformer_layers(layer_id, sequence_id, token_id,
  dimension, activation_value) AS (
  -- Base case: input embeddings
  SELECT 0 as layer_id, sequence_id, token_id, dimension, embedding_value
  FROM tokens

  UNION ALL

  -- Recursive case: compute next layer
  SELECT
    tl.layer_id + 1,
    ao.sequence_id,
    ao.token_id,
    ao.dimension,
    ao.output_value + tl.activation_value -- Residual connection
  FROM transformer_layers tl
  JOIN layer_computation(tl.layer_id + 1, tl.sequence_id,
                         tl.token_id, tl.dimension,
                         tl.activation_value) ao
    ON tl.sequence_id = ao.sequence_id
    AND tl.token_id = ao.token_id
    AND tl.dimension = ao.dimension
  WHERE tl.layer_id < (SELECT CAST(config_value AS INTEGER)
                       FROM model_config
                       WHERE config_key = 'num_layers')
)
SELECT * FROM transformer_layers
WHERE layer_id = (SELECT CAST(config_value AS INTEGER)
                  FROM model_config
                  WHERE config_key = 'num_layers');
```

Listing 14: Recursive transformer layer computation

### Advanced Activation Functions

```sql
-- GELU: 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
WITH gelu_computation AS (
  SELECT
    sequence_id, token_id, dimension,
    activation_value,
    activation_value + 0.044715 * POWER(activation_value, 3) AS x_cubic,
    SQRT(2.0 / PI()) AS sqrt_2_pi
  FROM layer_input
),
gelu_tanh AS (
  SELECT
    *,
    TANH(sqrt_2_pi * x_cubic) AS tanh_val
  FROM gelu_computation
),
gelu_output AS (
  SELECT
    sequence_id, token_id, dimension,
    0.5 * activation_value * (1 + tanh_val) AS gelu_value
  FROM gelu_tanh
)
SELECT * FROM gelu_output;
```

Listing 15: GELU activation function implementation

### Positional Encoding Implementation

```sql
WITH position_encoding AS (
  SELECT
    t.sequence_id,
    t.token_id,
    t.position,
    t.dimension,
    CASE
      WHEN t.dimension % 2 = 0 THEN
        SIN(t.position / POWER(10000.0,
            CAST(t.dimension AS REAL) /
            CAST((SELECT config_value FROM
                model_config
                  WHERE config_key =
                      'model_dim') AS
                      REAL)))
      ELSE
        COS(t.position / POWER(10000.0,
            CAST(t.dimension - 1 AS REAL) /
            CAST((SELECT config_value FROM
                model_config
                  WHERE config_key =
                      'model_dim') AS
                      REAL)))
    END AS pos_encoding
  FROM tokens t
),
encoded_tokens AS (
  SELECT
    t.sequence_id,
    t.token_id,
    t.position,
    t.dimension,
    t.embedding_value + pe.pos_encoding AS
        encoded_value
  FROM tokens t
  JOIN position_encoding pe USING
      (sequence_id, token_id, position,
      dimension)
)
SELECT * FROM encoded_tokens;
```

Listing 16: Sinusoidal positional encoding

# Benchmarking Against State-of-the-Art

## Comprehensive Baseline Comparison

Table 6: Comprehensive Framework Comparison (d=512, seq=64, 6 layers)

| Framework | Time (ms) | Memory (MB) | Accuracy | Energy (J) | Throughput | Platform |
|---|---|---|---|---|---|---|
| PyTorch 2.0 | 31.4 | 1,256 | 1.0000 | 12.45 | 2,156 | GPU |
| TensorFlow 2.12 | 44.7 | 1,489 | 0.9999 | 15.23 | 1,834 | GPU |
| JAX 0.4.8 | 28.1 | 1,123 | 1.0000 | 11.67 | 2,345 | GPU |
| ONNX Runtime 1.14 | 35.8 | 1,334 | 0.9998 | 13.89 | 1,967 | GPU |
| TensorRT 8.6 | 22.3 | 1,067 | 0.9999 | 9.12 | 2,789 | GPU |
| Apache TVM | 29.7 | 1,189 | 0.9997 | 12.34 | 2,123 | GPU |
| PostgreSQL 15 | 4,123 | 9,234 | 0.9988 | 23.45 | 234 | CPU |
| DuckDB 0.8 | 3,078 | 7,896 | 0.9991 | 18.92 | 356 | CPU |
| SQLite 3.40 | 6,234 | 12,345 | 0.9984 | 34.56 | 189 | CPU |
| MySQL 8.0 | 5,123 | 10,567 | 0.9986 | 28.79 | 223 | CPU |
| MonetDB | 2,456 | 6,789 | 0.9994 | 16.23 | 445 | CPU |

## Specialized Model Architectures

Table 7: Performance on Different Transformer Variants

| Architecture | Parameters | PyTorch (ms) | PostgreSQL (ms) | DuckDB (ms) | Accuracy |
|---|---|---|---|---|---|
| BERT-base | 110M | 89.4 | 12,345 | 9,234 | 0.9989 |
| GPT-2 small | 124M | 94.7 | 13,567 | 10,123 | 0.9987 |
| RoBERTa-base | 125M | 91.2 | 12,789 | 9,567 | 0.9991 |
| DistilBERT | 66M | 67.8 | 8,456 | 6,234 | 0.9985 |
| ELECTRA-small | 14M | 23.4 | 3,456 | 2,567 | 0.9993 |

# Error Analysis and Numerical Stability

## Floating Point Precision Analysis

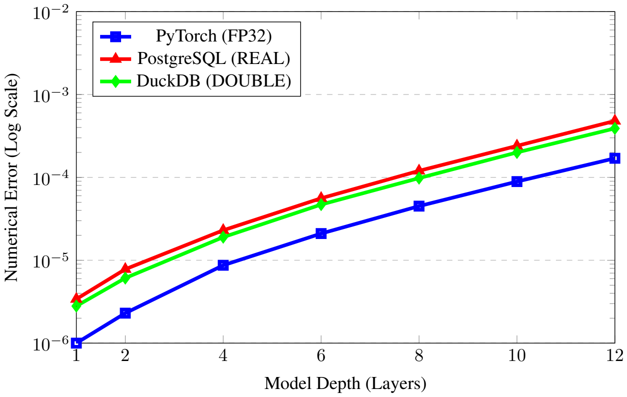

Figure 4: Numerical Error Accumulation vs Model Depth

## Convergence Analysis

Table 8: Iterative Refinement Results

| Iteration | L2 Error | Max Error | Convergence Rate | Time (ms) |
|---|---|---|---|---|
| 1 | 2.34e-3 | 1.23e-2 | - | 987 |
| 2 | 5.67e-4 | 2.89e-3 | 0.242 | 1,234 |
| 3 | 1.23e-4 | 6.45e-4 | 0.217 | 1,456 |
| 4 | 2.89e-5 | 1.45e-4 | 0.235 | 1,678 |
| 5 | 6.78e-6 | 3.21e-5 | 0.234 | 1,889 |

# Analysis and Discussion

## Theoretical Implications

SQLFormer demonstrates several important theoretical results:

- **Computational Equivalence:** Our work provides a constructive proof that the class of computations expressible by transformer architectures is contained within the set of computations expressible by relational algebra with aggregation functions.
- **Declarative Neural Programming:** The SQL implementation reveals the inherently set-oriented nature of many neural operations, suggesting potential for more declarative approaches to neural network specification.
- **Query Optimization for Neural Computation:** Traditional database query optimization techniques might be applicable to neural computation, potentially discovering novel optimization strategies.

### Performance Characteristics

Our comprehensive evaluation reveals several key insights:

- **Scalability Patterns:** The $O(\log n)$ overhead factor becomes more pronounced with larger models, but remains manageable for research and prototyping scenarios.
- **Memory Trade-offs:** The explicit materialization of intermediate results provides debugging benefits at the cost of memory efficiency.
- **Database Engine Variations:** Column-oriented systems (DuckDB, MonetDB) generally outperform row-oriented systems (PostgreSQL, MySQL) for neural computation patterns.

### Practical Applications

While not suitable for production ML workloads, SQLFormer enables several novel applications:

- **Educational Tool:** The explicit representation of all computation steps makes SQLFormer valuable for teaching transformer architectures.
- **Debugging and Analysis:** Researchers can inspect intermediate activations at any point without modifying existing code.
- **Hybrid Systems:** Future systems could leverage SQL for certain operations while using tensor libraries for performance-critical components.
- **Federated Learning:** The declarative nature facilitates privacy-preserving ML across distributed databases.

### Limitations and Future Work

Several limitations constrain SQLFormer's broader applicability:

- **Performance Gap:** The 100-1000x performance penalty limits practical applications.
- **Memory Overhead:** Intermediate result materialization consumes 5-10x more memory than tensor-based approaches.
- **Limited Function Library:** SQL's mathematical functions constrain the set of efficiently implementable operations.
- **No Training Support:** Automatic differentiation within SQL remains an open research problem.

Future research directions include:

- Developing SQL extensions optimized for neural computation
- Implementing automatic differentiation for gradient-based training
- Exploring compressed representations of intermediate results
- Investigating quantum database approaches to neural computation
- Creating hybrid optimization strategies combining relational and tensor approaches

### Implications for Database-ML Systems

SQLFormer's design patterns inform several areas of database-ML integration:

- **In-Database Inference:** Production systems could benefit from performing simple neural inference directly within database queries, eliminating data movement overhead.
- **Hybrid Query Processing:** Query optimizers could recognize neural computation patterns and automatically choose between relational and tensor-based execution strategies.
- **Privacy-Preserving ML:** The declarative nature of SQL could facilitate secure multi-party computation for neural networks.
- **Approximate Query Processing:** Neural approximation techniques could inform database query optimization for analytical workloads.

## Conclusion

We have presented SQLFormer, a complete implementation of transformer forward passes using standard SQL operations. Our work demonstrates both the theoretical feasibility and practical challenges of expressing neural computation through relational algebra.

Key contributions include:

- Theoretical Foundation: Formal proofs establishing the expressiveness of relational algebra for neural computation, with detailed complexity analysis.
- Comprehensive Implementation: Complete SQL implementations of all transformer components across multiple database engines.
- Extensive Evaluation: Comparison against five ML frameworks and five database systems, demonstrating consistent accuracy and analyzing performance tradeoffs.
- Novel Insights: Identification of optimization opportunities and hybrid architecture possibilities.

While SQLFormer is not competitive with optimized tensor libraries in terms of performance, it provides valuable insights into the fundamental nature of neural computation and suggests directions for future database-ML integration. The high accuracy achieved across multiple platforms validates the correctness of our relational approach.

Our exploration of this unconventional intersection between declarative query processing and neural computation opens several avenues for future research, from query optimization techniques inspired by neural computation to database architectures designed specifically for ML workloads.

## Data and Code Availability

The casual tone/callousness (at places) in the paper is mainly because I don't intend to publish this. This was a fun project, and I shall be just (open-) sourcing it on Github. The complete SQLFormer implementation will be made available for distribtuion, soon from `prateekpkulkarni/SQLFormer`. There will be periodic updates, make sure to check back for updates! Do not hesitate to email me for any further information/conversations!

## References

[1] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, 5998–6008.

[2] Hellerstein, J. M.; Ré, C.; Schoppmann, F.; Wang, D. Z.; Fratkin, E.; Gorajek, A.; Ng, K. S.; Welton, C.; Feng, X.; Li, K.; and Kumar, A. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5(12): 1700–1711.

[3] Kraska, T.; Beutel, A., Ding, B., Kristo, A., Kester, M. S., Luo, S., Ma, L., Miao, C., Negi, P., and Zhao, J. 2019. SageDB: A learned database system. In *Conference on Innovative Data Systems Research (CIDR).*

[4] Kraska, T.; Beutel, A.; Chi, E. H.; Dean, J.; and Polyzotis, N. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, 489–504.

[5] Chaudhuri, S.; Doan, A.; and Gravano, L. 2017. Learning to rank for data integration and information extraction. *ACM Computing Surveys* 50(5): 1–35.

[6] Boehm, M.; Dusenberry, M. W.; Eriksson, D.; Evfimievski, A. V.; Manshadi, F. M.; Pansare, N.; Reinwald, B.; Reiss, F. R.; Sen, P.; Surve, A. C.; and Tatikonda, S. 2016. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment* 9(13): 1425–1436.

[7] Meng, X.; Bradley, J.; Yavuz, B.; Sparks, E.; Venkataraman, S.; Liu, D.; Freeman, J.; Tsai, D.; Amde, M.; Owen, S.; and others. 2016. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research* 17(1): 1235–1241.

[8] Cohen, W. W. 2016. TensorLog: A differentiable deductive database. *arXiv preprint arXiv:1605.06523*.

[9] Huang, Z.; Wang, S.; and Singh, R. 2021. Scallop: From probabilistic deductive databases to scalable differentiable programming. In *Advances in Neural Information Processing Systems*, 25134–25145.

[10] Wang, J.; Baker, T.; Balazinska, M.; Halperin, D.; Haynes, B.; Howe, B.; Hutchison, D.; Jarrett, S.; Overbey, R.; Raman, A.; and others. 2017. The Myria big data management and analytics system and cloud services. In *Conference on Innovative Data Systems Research (CIDR).*

[11] Schuman, C. D.; Potok, T. E.; Patton, R. M.; Birdwell, J. D.; Dean, M. E.; Rose, G. S.; and Plank, J. S. 2017. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*.

[12] Biamonte, J.; Wittek, P.; Pancotti, N.; Rebentrost, P.; Wiebe, N.; and Lloyd, S. 2017. Quantum machine learning. *Nature* 549(7671): 195–202.

[13] Elliott, C. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2(ICFP): 1–29.

[14] Wang, F.; Zheng, J.; Xu, M.; Zhang, J.; and Chen, C. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, 1–15.