Databases as AI Runtimes

Rihan Hai

Agenda

1 Transformer Architecture

Core components, attention mechanisms, and mathematical foundations, example

2 LLaMA Architecture

Architectural innovations and key differences from standard Transformers

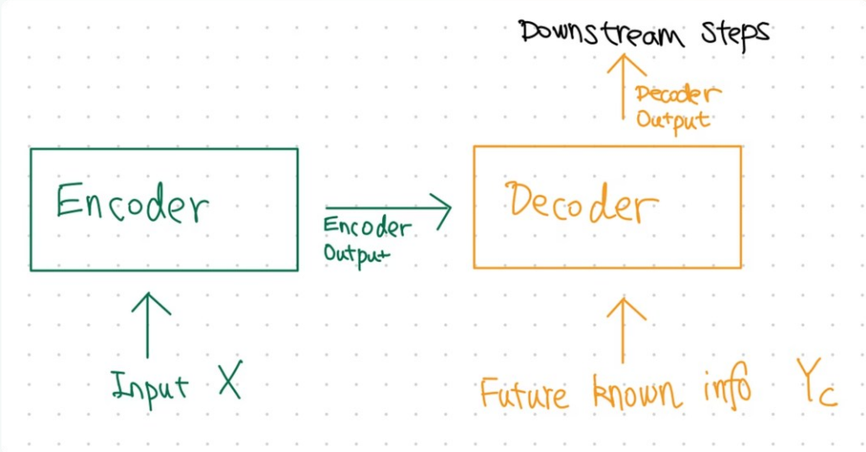
3 Mixture of Experts

Traditional MoE in transformer, DeepSeekMoE

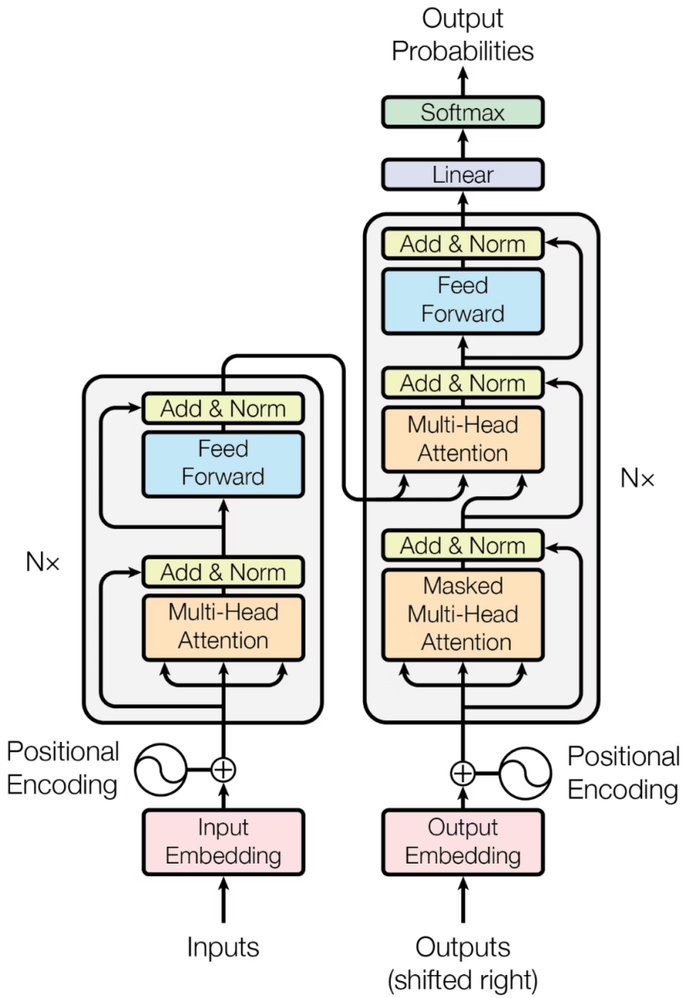
4 Databases as AI Runtimes

Serving Large Language Models with SQL on Low-Resource Hardware

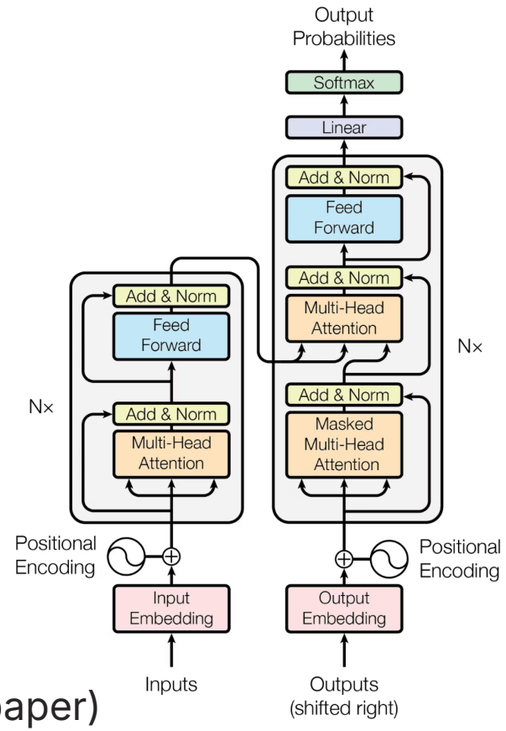
Part I: Transformer Architecture



Transformer Architecture



Transformer Architecture: Overview



Encoder Stack:

- N identical layers (N=6 in original paper)
- Each layer has two sub-layers:
  - Multi-head self-attention mechanism
  - Position-wise fully connected feed-forward network
- Residual connection around each sub-layer
- Layer normalization after each residual connection

Decoder Stack:

- N identical layers (N=6 in original paper)
- Each layer has three sub-layers:
  - Masked multi-head self-attention
  - Multi-head attention over encoder output
  - Position-wise feed-forward network
- Residual connections and layer normalization

Key Components of Transformer

1

Embedding Layers

Convert tokens to dense vector representations of dimension dmodel (512 in original)

2

Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

- pos: The position of the token in the sequence.
- i: The dimension index (from 0 to $d_{model}/2 - 1$).
- d_{model}: The dimension of the embedding.

3

Multi-Head Attention

Allows model to jointly attend to information from different representation subspaces

4

Feed-Forward Networks

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

5

Residual Connections & Layer Normalization

$$x + Sublayer(x)$$

$$LayerNorm(x + Sublayer(x))$$

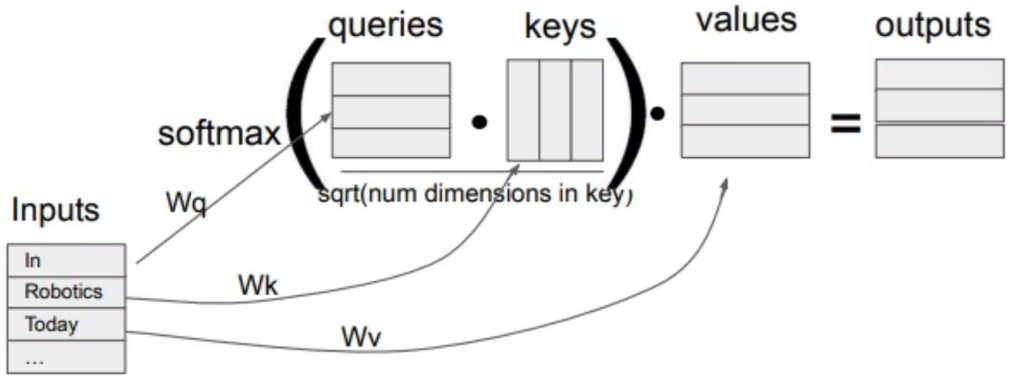
Scaled Dot-Product Attention

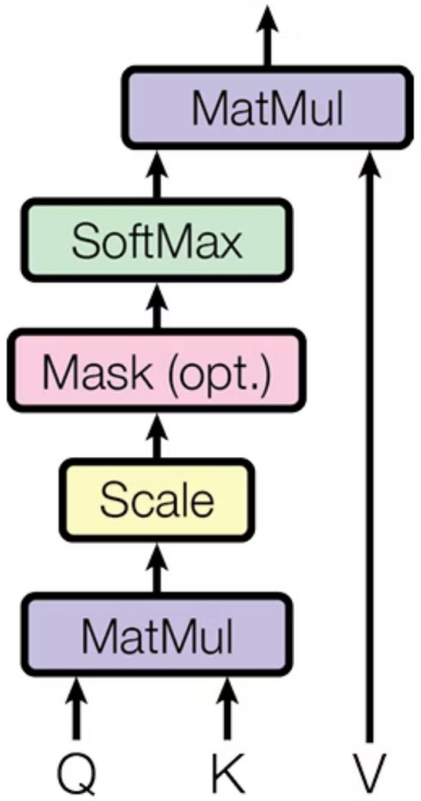
The basic attention mechanism in Transformers:

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

Where:

- Q: Query matrix of shape. Represents the current element being processed.
- K: Keys matrix of shape. Represents all elements in the sequence that the query might attend to.
- V: Values matrix of shape. Contains the actual information associated with each key.
- d_k: Model dimension (the dimension of the keys). Used for scaling to prevent the dot products from becoming too large and pushing the softmax into regions with extremely small gradients.





10

Self-Attention: Step-by-Step

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

1

Input Embeddings

For a sentence with tokens $x_1, x_2, ..., x_n$, each token is embedded into a vector of dimension d_{model}

2

Linear Projections

Project each embedding to create queries, keys, and values:

$$Q = XW^Q$$
$$K = XW^K$$
$$V = XW^V$$

3

Attention Scores

Compute attention scores with scaling to prevent vanishing gradients:

$$S = \frac{QK^T}{\sqrt{d_k}}$$

4

Attention Weights

Apply softmax to get attention weights:

$$A = \text{softmax}(S)$$

5

Weighted Values

Compute weighted sum of values:

$$Z = AV$$

Multi-Head Attention

Instead of performing a single attention function, the Transformer uses multiple attention heads in parallel:

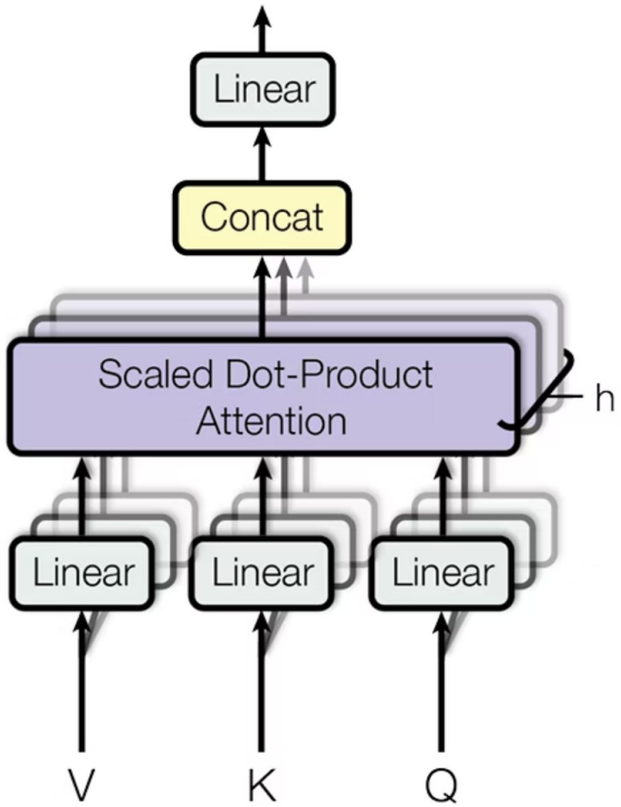
$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Where:

- W_i^Q, W_i^K, W_i^V: Linear projections for queries, keys, and values
- W^O: Output projection matrix

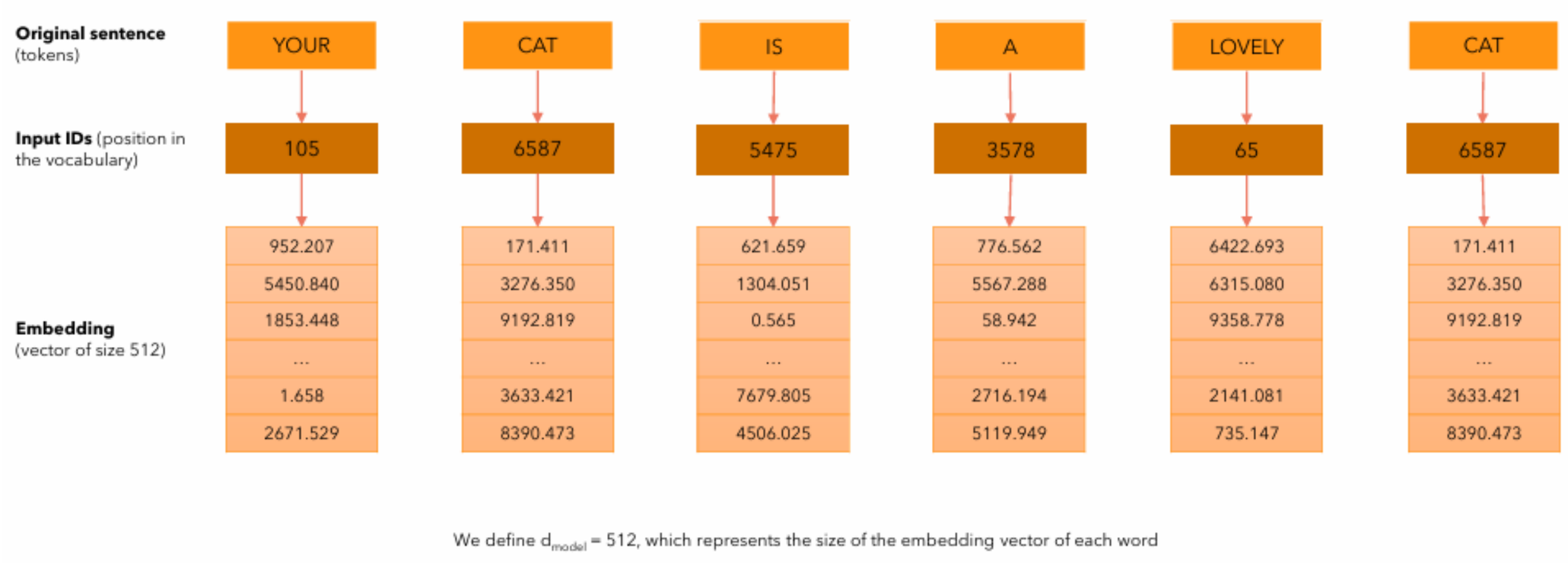
In the original paper: h = 8, d_{model} = 512, $d_k = d_v = \frac{d_{model}}{h}$ = 64



12

Example

Input embedding



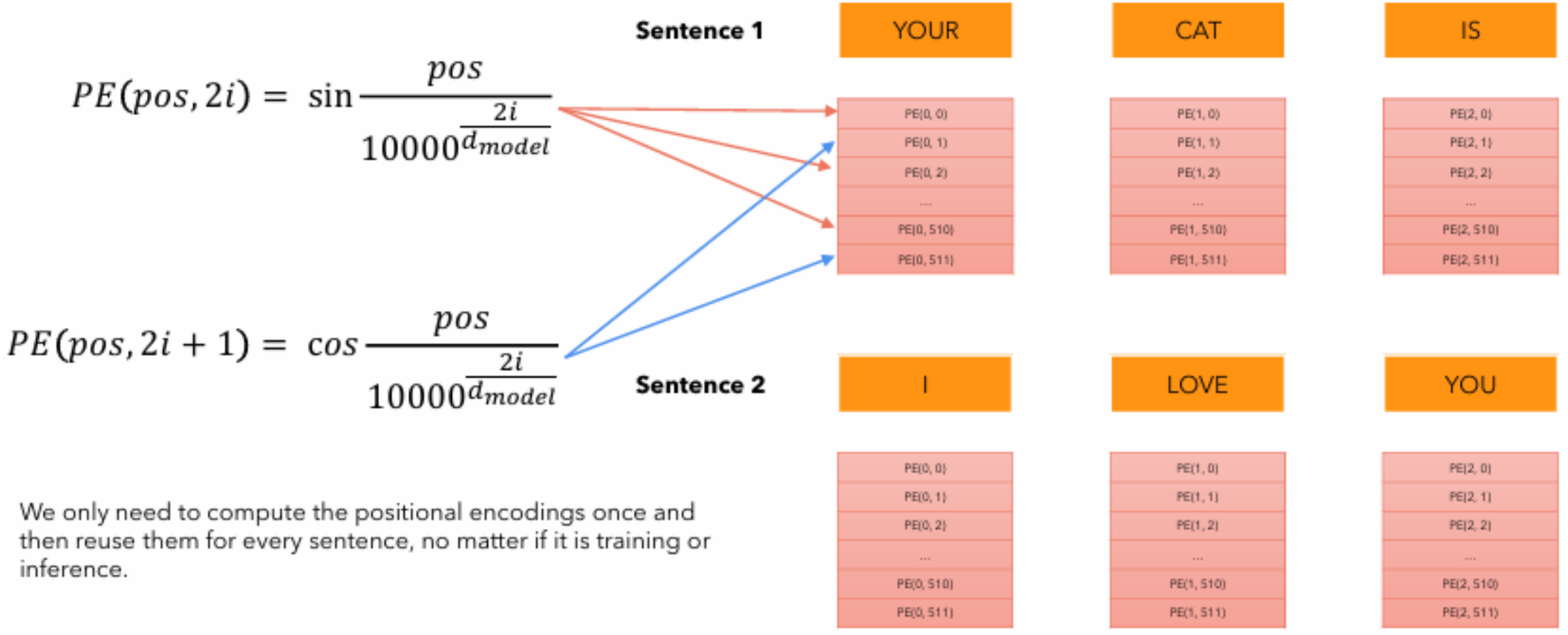
We define $d_{model} = 512$, which represents the size of the embedding vector of each word

Example

Positional encoding

Example

Positional encoding



Example

Self-Attention

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d_{model}} = \mathbf{d_k} = 512$.

The matrices **Q**, **K** and **V** are just the input sentence.

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

$$\text{softmax}\left(\frac{\mathbf{Q}\ (6, 512) \times \mathbf{K^T}\ (512, 6)}{\sqrt{512}}\right) =$$

	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

(6, 6)

* all values are random.

* for simplicity I considered only one head, which makes $d_{model} = d_k$.

Example

Self-Attention

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

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
A	0.210	0.128	0.206	0.212	0.119	0.125
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174
CAT	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6)

X

V

(6, 512)

=

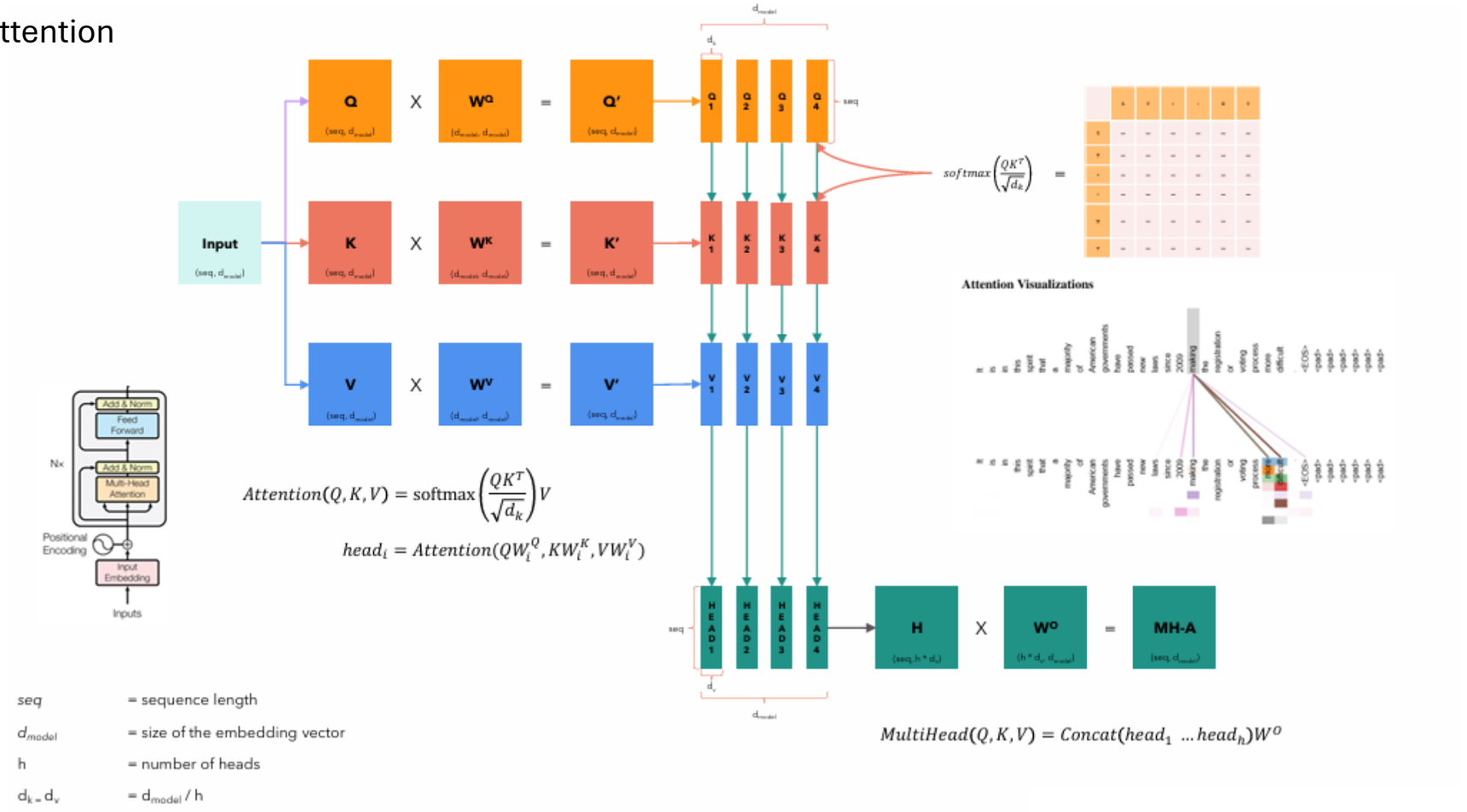
Attention

(6, 512)

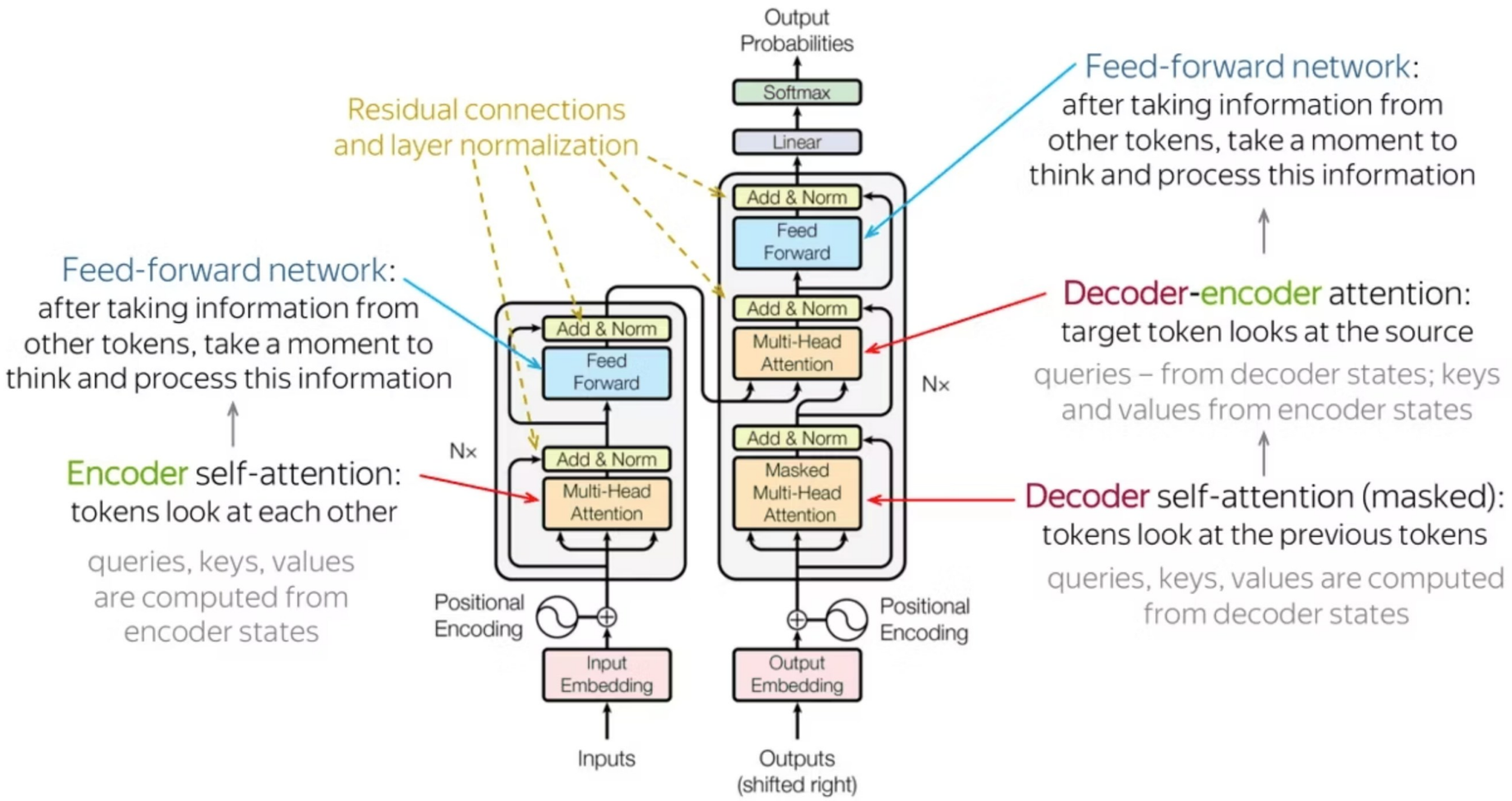
Each row in this matrix captures not only the meaning (given by the embedding) or the position in the sentence (represented by the positional encodings) but also each word's interaction with other words.

Example

Multi-head Attention



Summary: Transformer Architecture



Part II: LLaMA Architecture

Evolution: Llama 1 → Llama 2 → Llama 3

1

Llama 1 (2023)

- Base transformer with RMSNorm
- SwiGLU activations
- Rotary positional embeddings (RoPE)
- 7B to 65B parameters
- Context length: 2048 tokens

2

Llama 2 (2023)

- Grouped-query attention (GQA)
- Improved pretraining mixture
- 7B to 70B parameters
- Context length: 4096 tokens
- Reinforcement Learning from Human Feedback alignment

3

Llama 3 (2024)

- Advanced tokenizer (128K vocab)
- 8B, 70B, 405B parameters
- Context length: up to 8K tokens (for released models)
- High-quality pretraining data (15T tokens)
- Post-training: Supervised Finetuning + Reinforcement Learning from Human Feedback alignment

Architectural Innovations

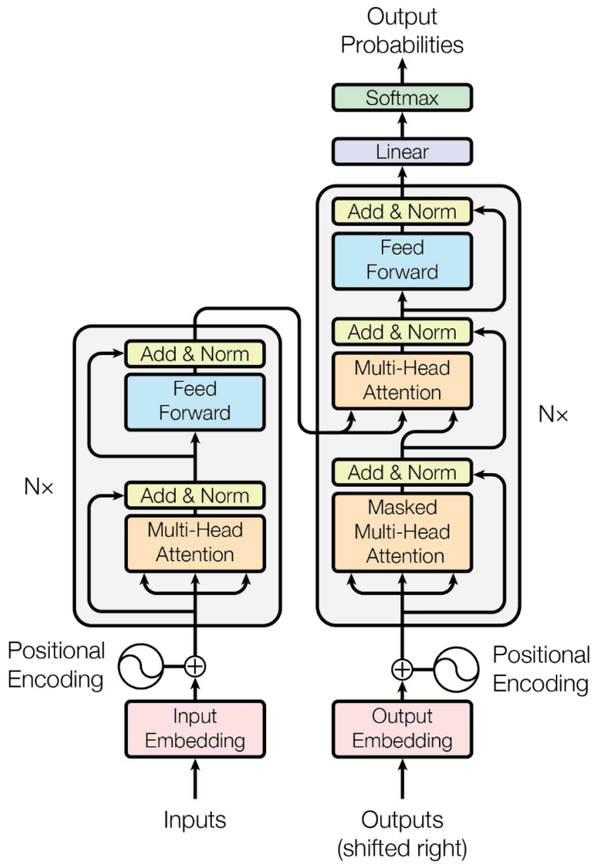
Consistent transformer core (RMSNorm, SwiGLU, RoPE). Llama-2 adds GQA (notably at 70B). Llama-3 focuses on tokenization efficiency and training-data scale for better parameter efficiency.

Training Methodology

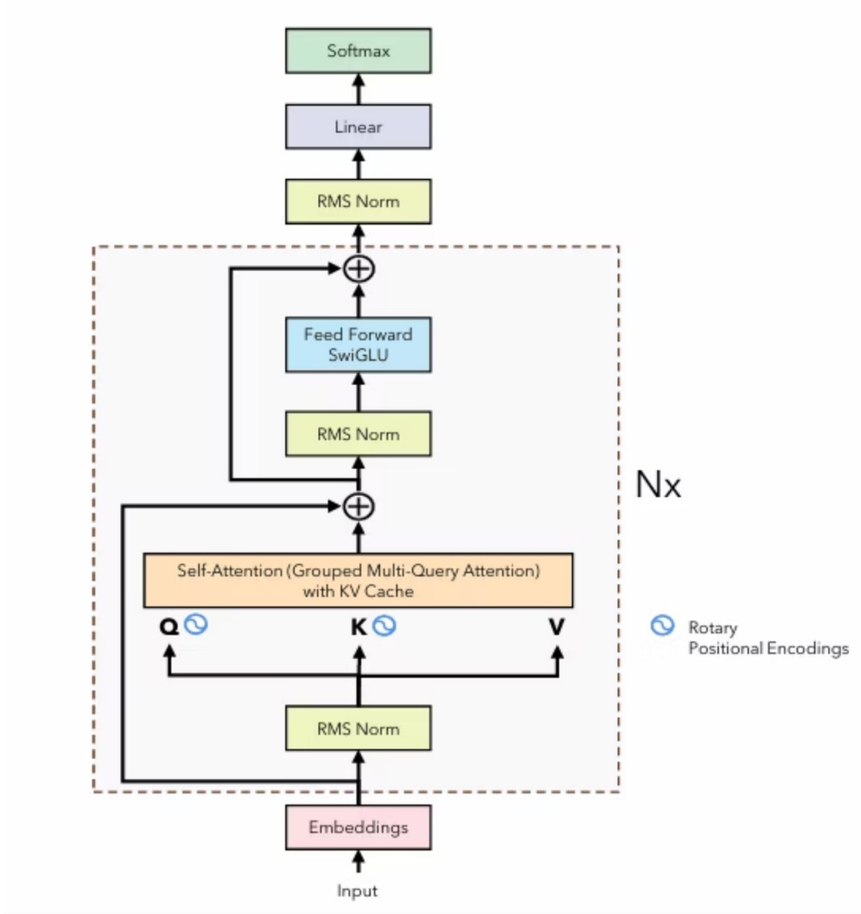
All generations use causal next-token pretraining. Llama-2 and Llama-3 add instruction tuning and preference-based alignment (e.g., RLHF). Llama-3 scales pretraining to ~15T tokens and introduces a more efficient 128K-vocab tokenizer.

Dubey, Abhimanyu, et al. "The llama 3 herd of models." arXiv e-prints (2024): arXiv-2407.

LLaMA: Architecture Evolution



Transformer



LLaMA

Llama: Architecture Evolution

Key Architectural Modifications

Meta's Llama models advance the transformer architecture with these key modifications:

Grouped Multi-Query Attention (GQA)

A hybrid attention mechanism balancing Multi-Head Attention's quality with Multi-Query Attention's speed by grouping heads for shared Key/Value projections, reducing memory and improving inference.

Rotary Position Embeddings (RoPE)

Encodes absolute positional information with relative attention bias:

$$R_{\Theta,m,2i}(x) = x_{2i}\cos(m\theta_i) - x_{2i+1}\sin(m\theta_i)$$

$$R_{\Theta,m,2i+1}(x) = x_{2i}\sin(m\theta_i) + x_{2i+1}\cos(m\theta_i)$$

Efficiency Optimizations

KV Cache Optimization

Stores pre-computed Key (K) and Value (V) projections of previous tokens to speed up autoregressive decoding, significantly reducing re-computation and memory bandwidth during inference.

Pre-normalization

Uses RMSNorm before attention and FFN, rather than post-normalization:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} \cdot \gamma$$

SwiGLU Activation

Replaces ReLU with SwiGLU in feed-forward networks:

$$\text{SwiGLU}(x) = x \cdot \sigma(W_1 x) \cdot W_2 x$$

Where σ is the sigmoid function

Grouped Multi-Query Attention (GQA)

Key Concepts

- **Number of Query Heads (H_q):** The total number of distinct query projections.
- **Number of Key/Value Heads (H_{kv}):** The number of shared Key and Value projections, where $H_{kv} < H_q$.
- **Group Size (G):** The number of query heads that share a single Key and Value head, calculated as $G = H_q / H_{kv}$.

Grouped Multi-Query Attention (GQA)

The attention mechanism for each query head h_q still follows the standard scaled dot-product attention. However, for a given query head Q_{h_q}, its corresponding Key $K_{h_{kv}}$ and Value $V_{h_{kv}}$ projections are shared with *G-1* other query heads within its group. Specifically, if a query head *h_q* belongs to group *i*, it will use the *i*-th shared Key and Value heads, K_i and V_i

$$\text{Attention}(Q_{h_q}, K_{h_{kv}}, V_{h_{kv}}) = \text{softmax}\left(\frac{Q_{h_q} K_{h_{kv}}^T}{\sqrt{d_k}}\right) V_{h_{kv}}$$

Where:

- $Q_{h_q} \in \mathbb{R}^{L \times d_k}$ is the query matrix for query head h_q.
- $K_{h_{kv}} \in \mathbb{R}^{L \times d_k}$, $V_{h_{kv}} \in \mathbb{R}^{L \times d_v}$ are the key matrix and value matrix for the H_{kv} group that h_q belongs to.
- L is the sequence length; d_k is the dimension of keys and queries; d_v is the dimension of values.

The outputs from all H_{q} attention heads are then concatenated and linearly projected, similar to Multi-Head Attention.

Why KV Cache: Example

Next Token Prediction Task: Inference

Output Love

Inference
T = 1

Input [SOS]

Why KV Cache: Example

Next Token Prediction Task: Inference

Output Love that

Inference
T = 2

Input [SOS] Love

Why KV Cache: Example

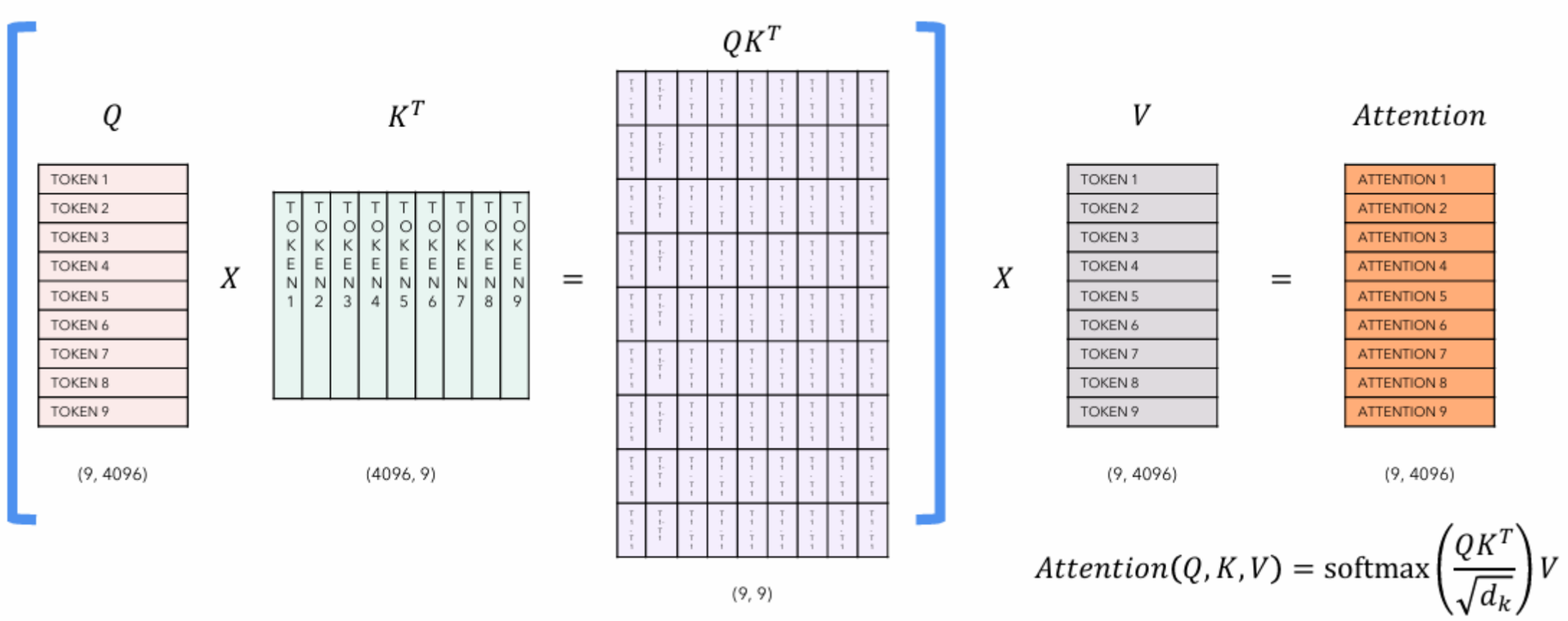
Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle heart [EOS]

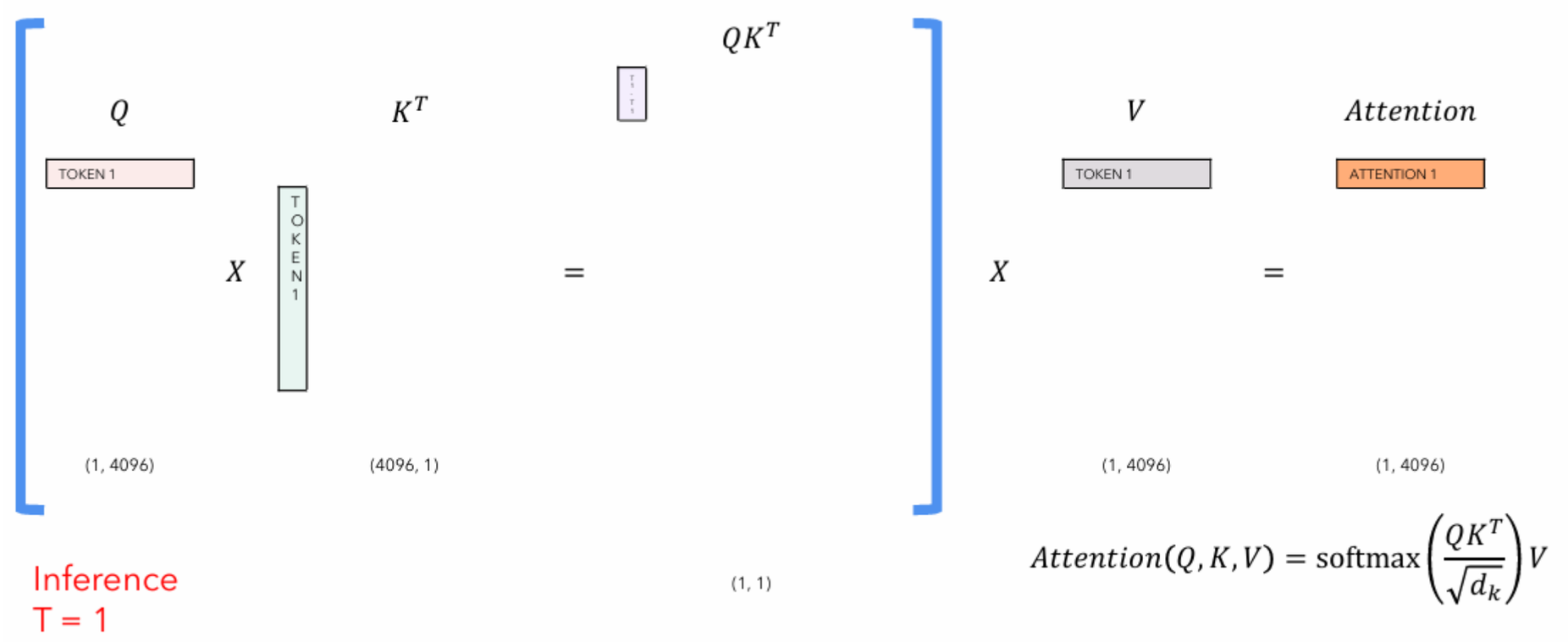
Inference
T = 9

Input [SOS] Love that can quickly seize the gentle heart

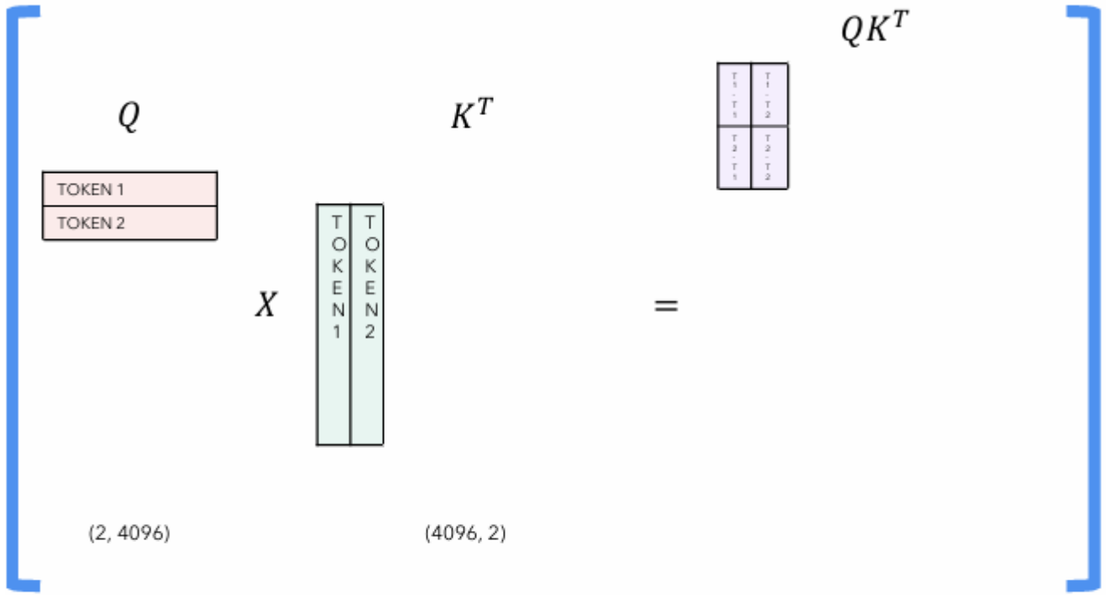
Why KV Cache: Example

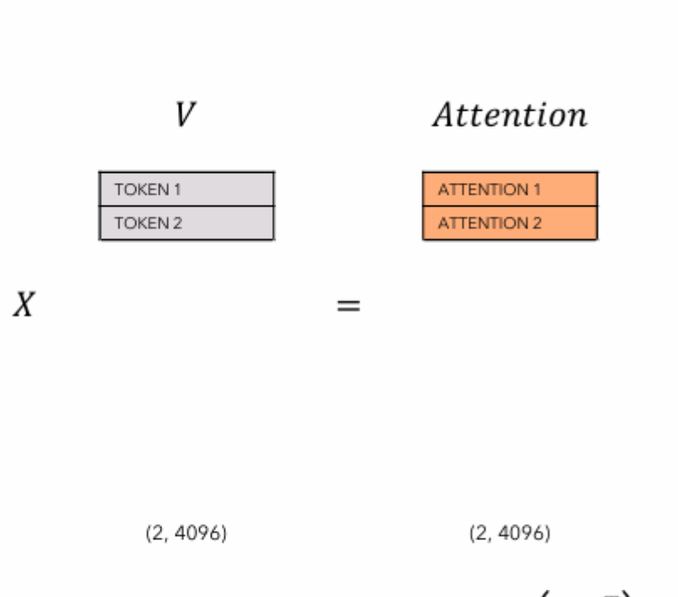


Why KV Cache: Example



Self-Attention during Next Token Prediction Task

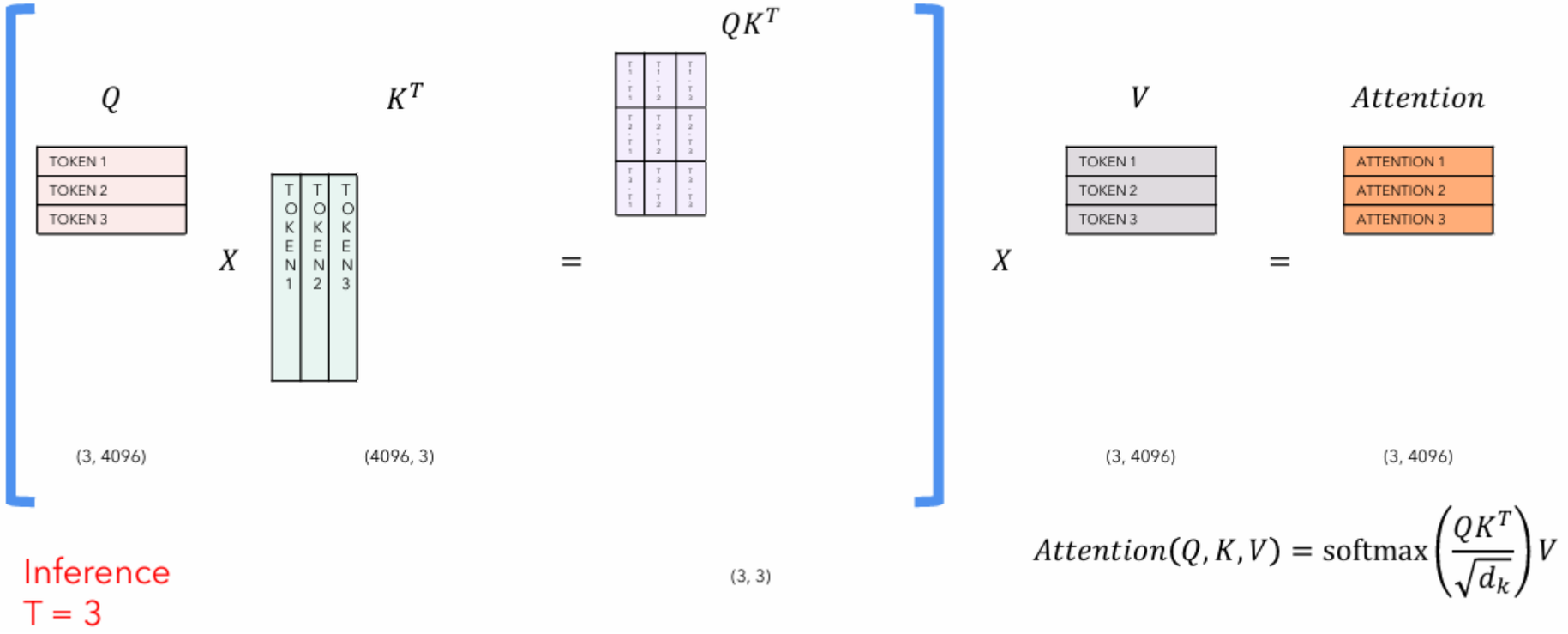




Inference
T = 2

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

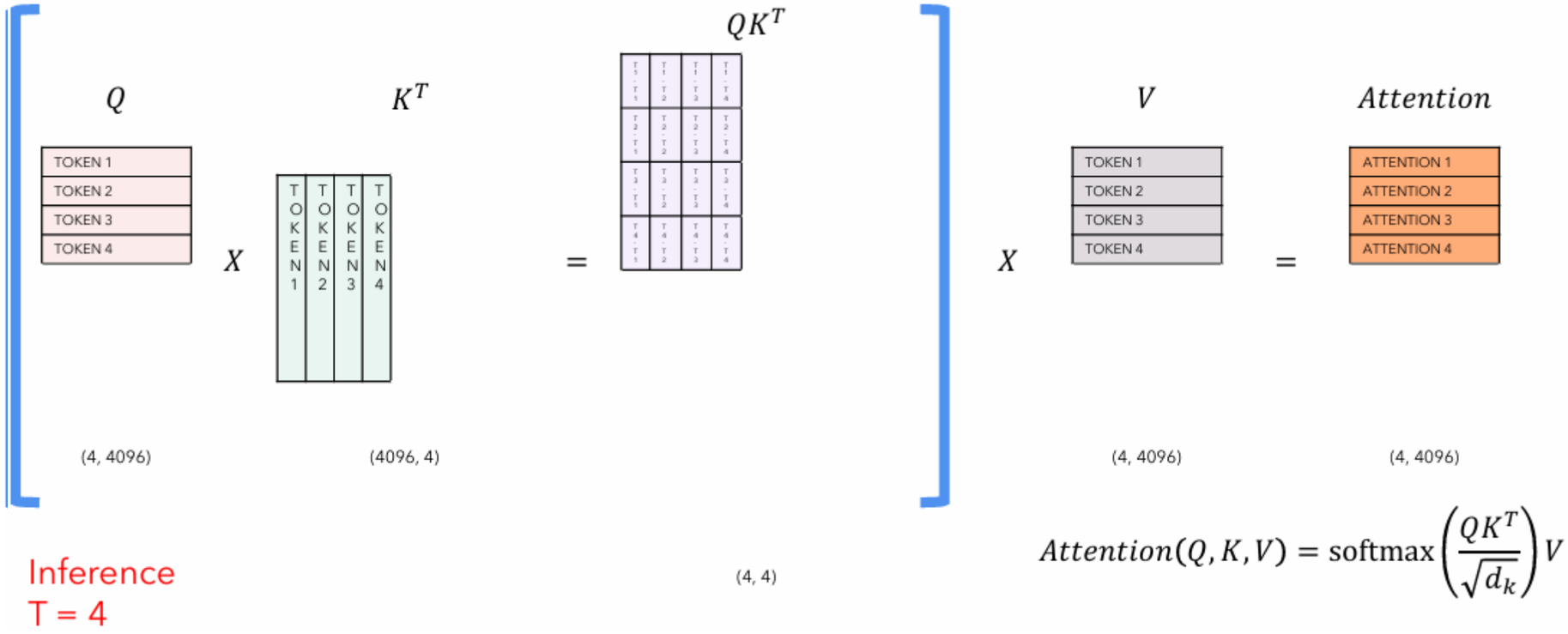
Self-Attention during Next Token Prediction Task



Inference
T = 3

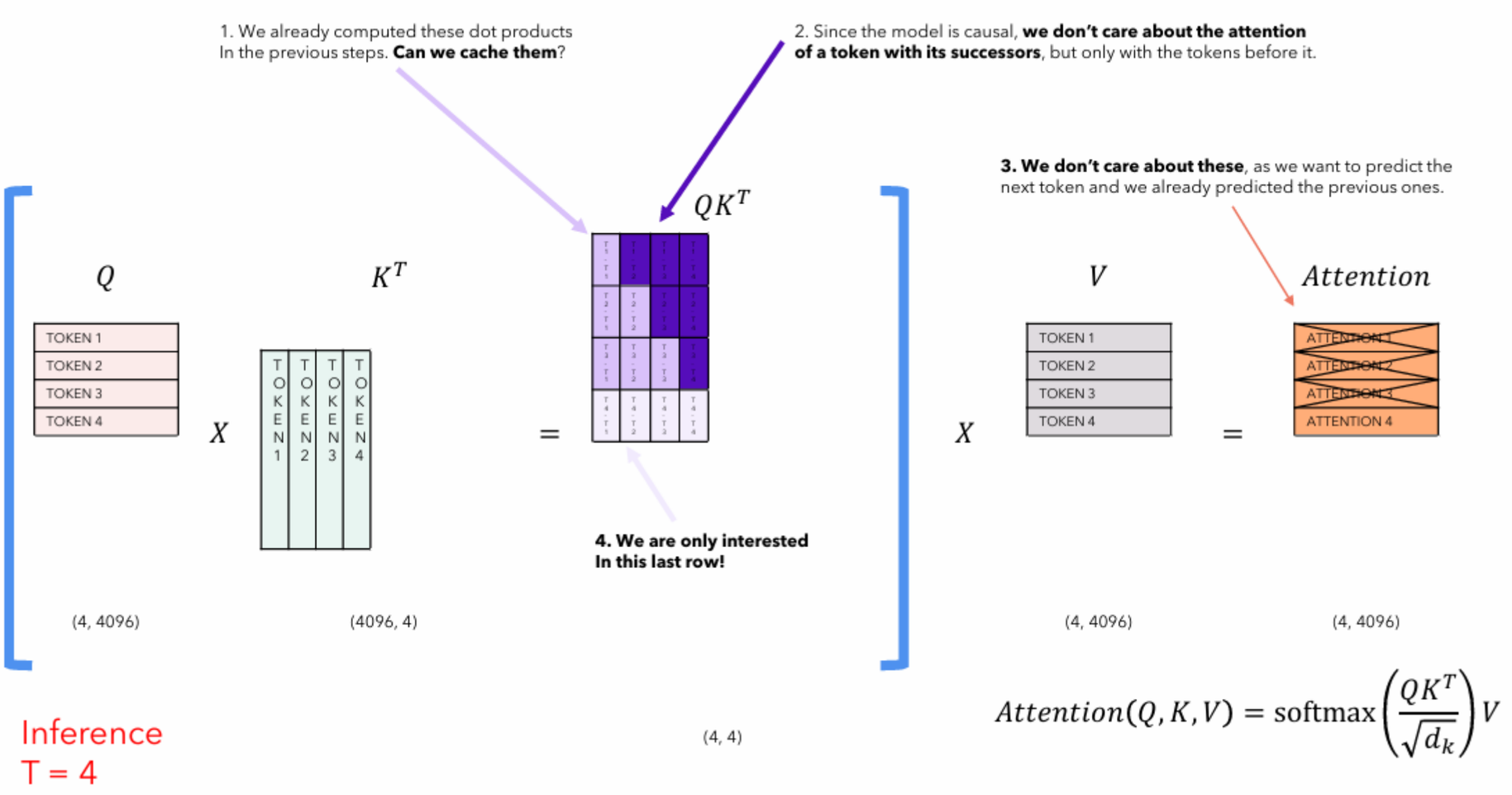
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention during Next Token Prediction Task



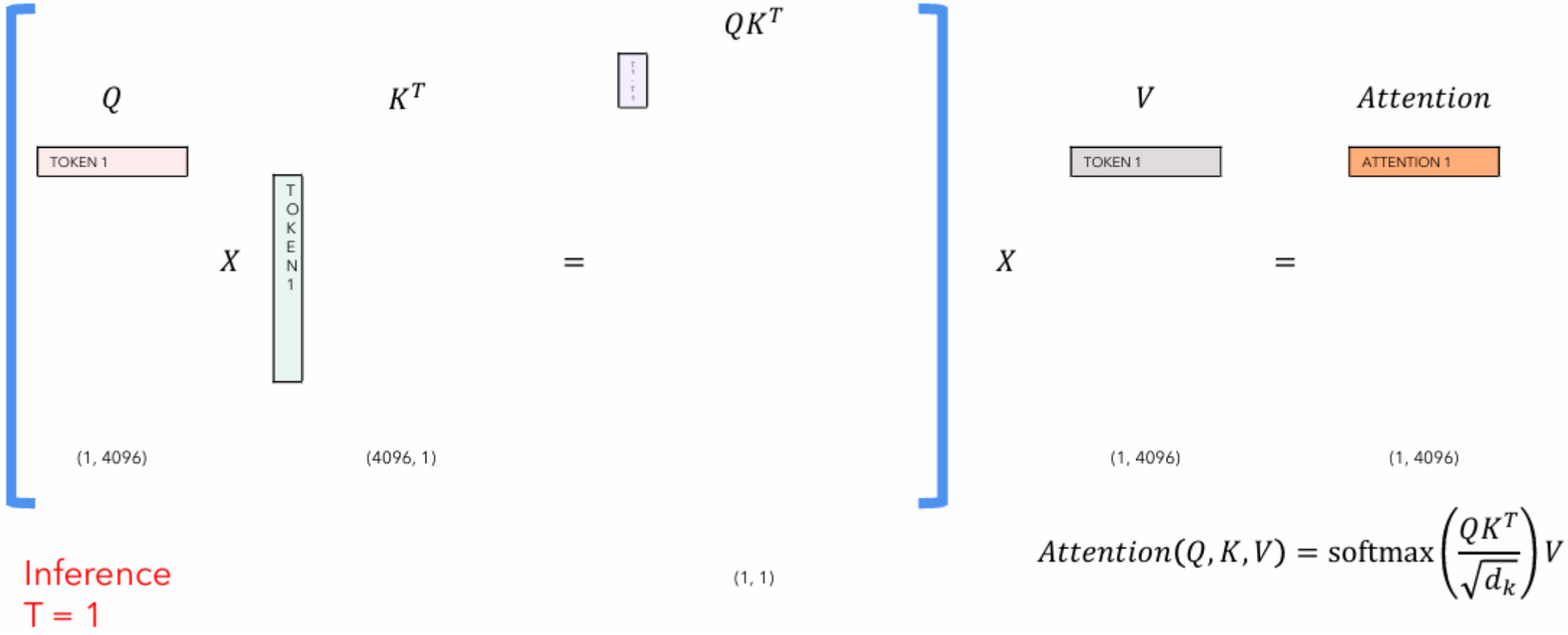
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention during Next Token Prediction Task



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

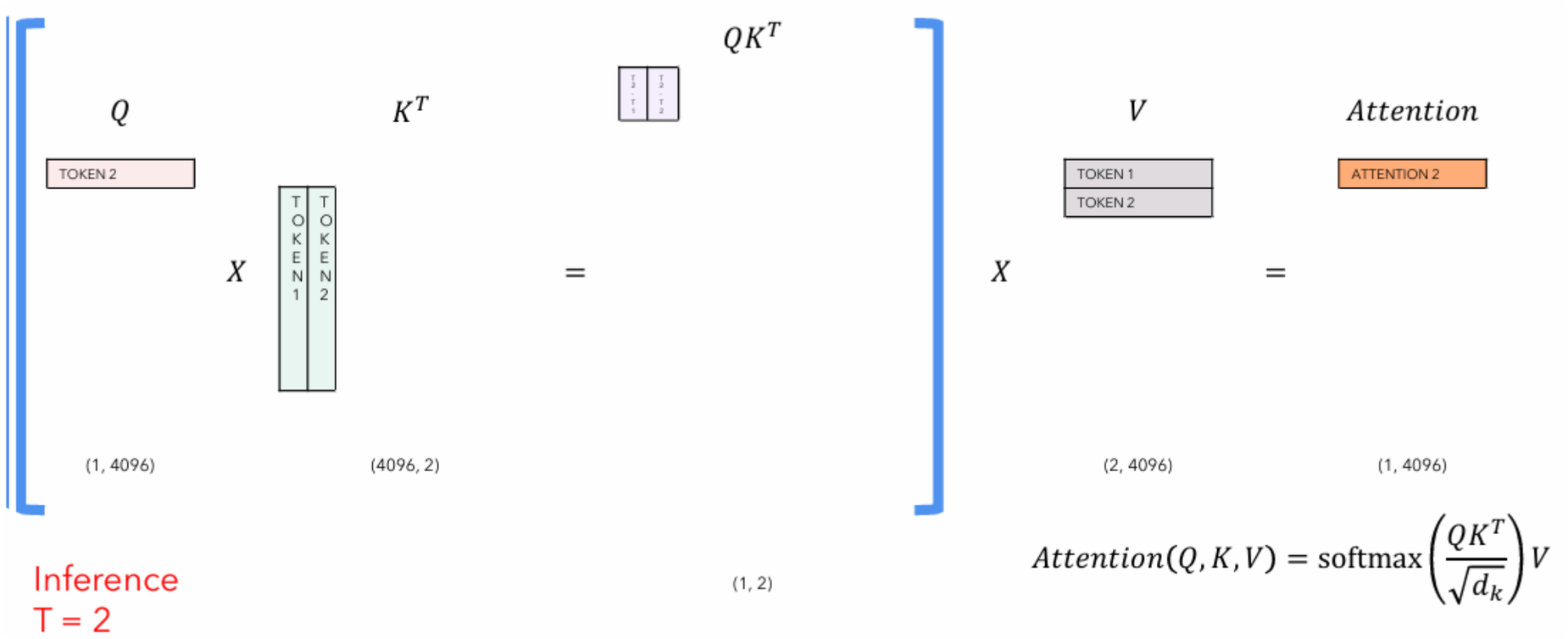
Self-Attention with KV-Cache



Inference
T = 1

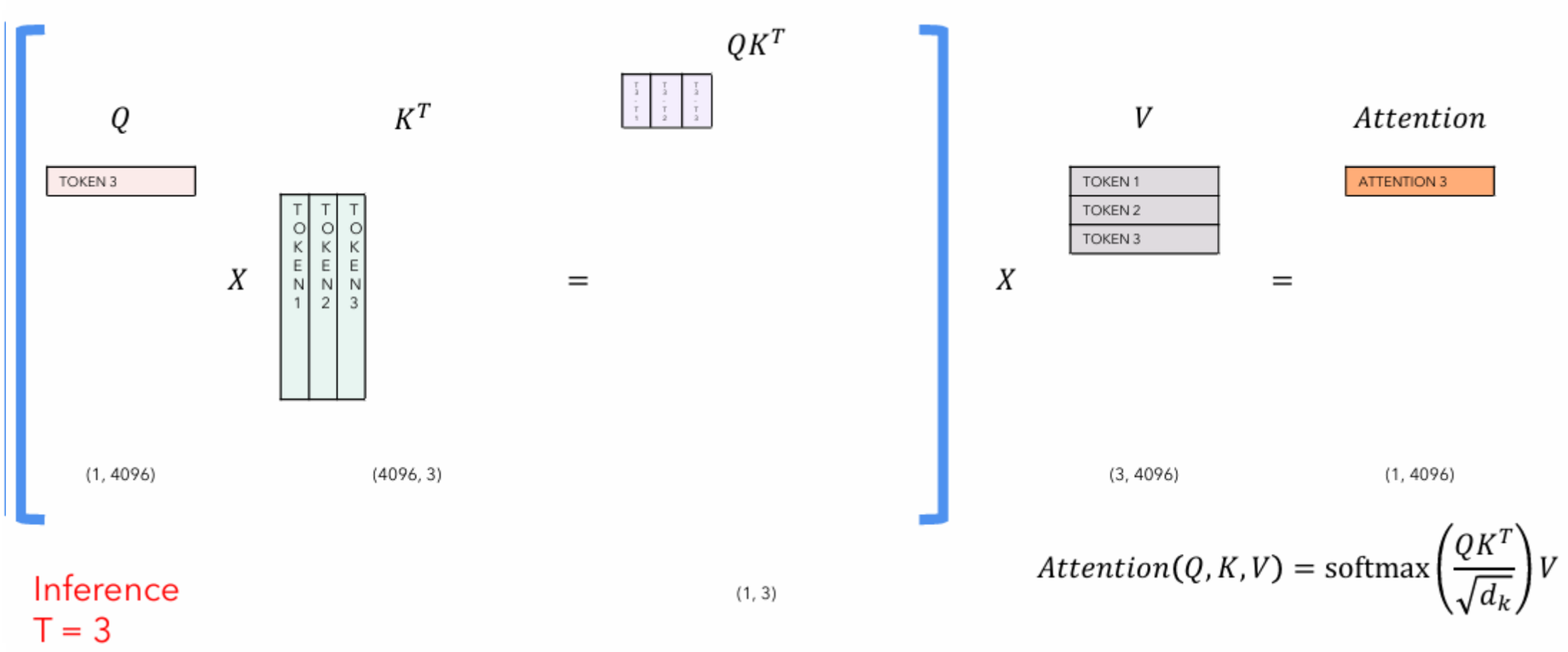
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention with KV-Cache

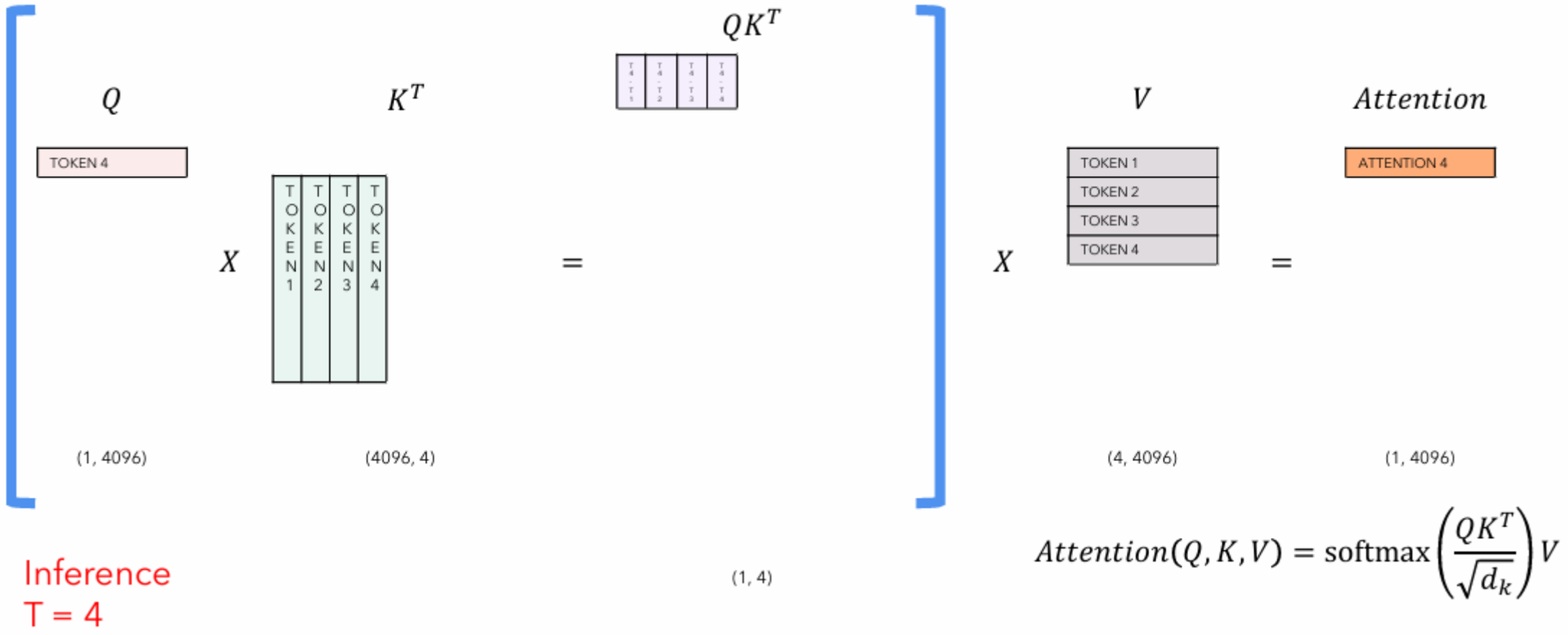


$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention with KV-Cache



Self-Attention with KV-Cache



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

Grouped Multi-Query Attention

Multi-Head Attention

- High quality
- Computationally slow

Grouped Multi-Query Attention

- A good compromise between quality and speed

Multi-Query Attention

- Loss in quality
- Computationally fast

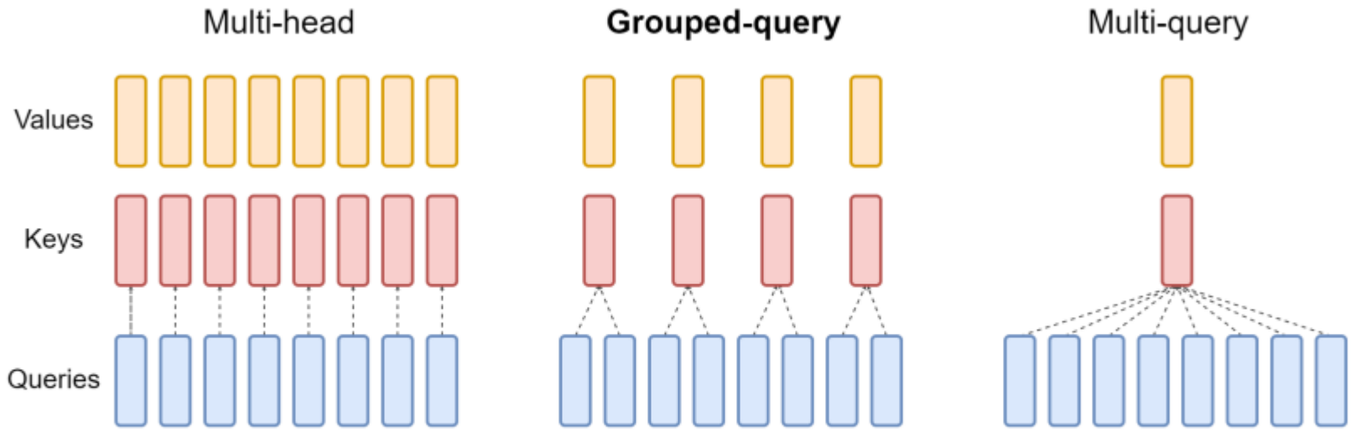


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Joshua Ainslie et al. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints https://arxiv.org/abs/2305.13245

SwiGLU Activation in Feed-Forward Networks

Standard Transformer FFN:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

Uses ReLU activation

LLaMA SwiGLU FFN:

$$FFN_{SwiGLU}(x, W, V, W_2) = (Swish_1(xW) \otimes xV)W_2$$

Uses SwiGLU activation; offers better gradient flow and performance

SwiGLU: Mathematical Details

The SwiGLU activation function in LLaMA is defined as:

$$SwiGLU(x, W, V, b, c, \beta) = (Swish_\beta(xW + b) \otimes (xV + c)$$

⊗ denotes element-wise multiplication

Often we omit bias terms and choose Swish function with *β=1*, i.e., Sigmoid Linear Unit (SiLU) function: $\text{Swish}(x) = x \cdot \sigma(\beta x) = \frac{x}{1+e^{-\beta x}}$

$$FFN_{SwiGLU}(x, W, V, W_2) = (Swish_1(xW) \otimes xV)W_2$$

Compared to standard FFN: fewer parameters & maintaining or improving performance.

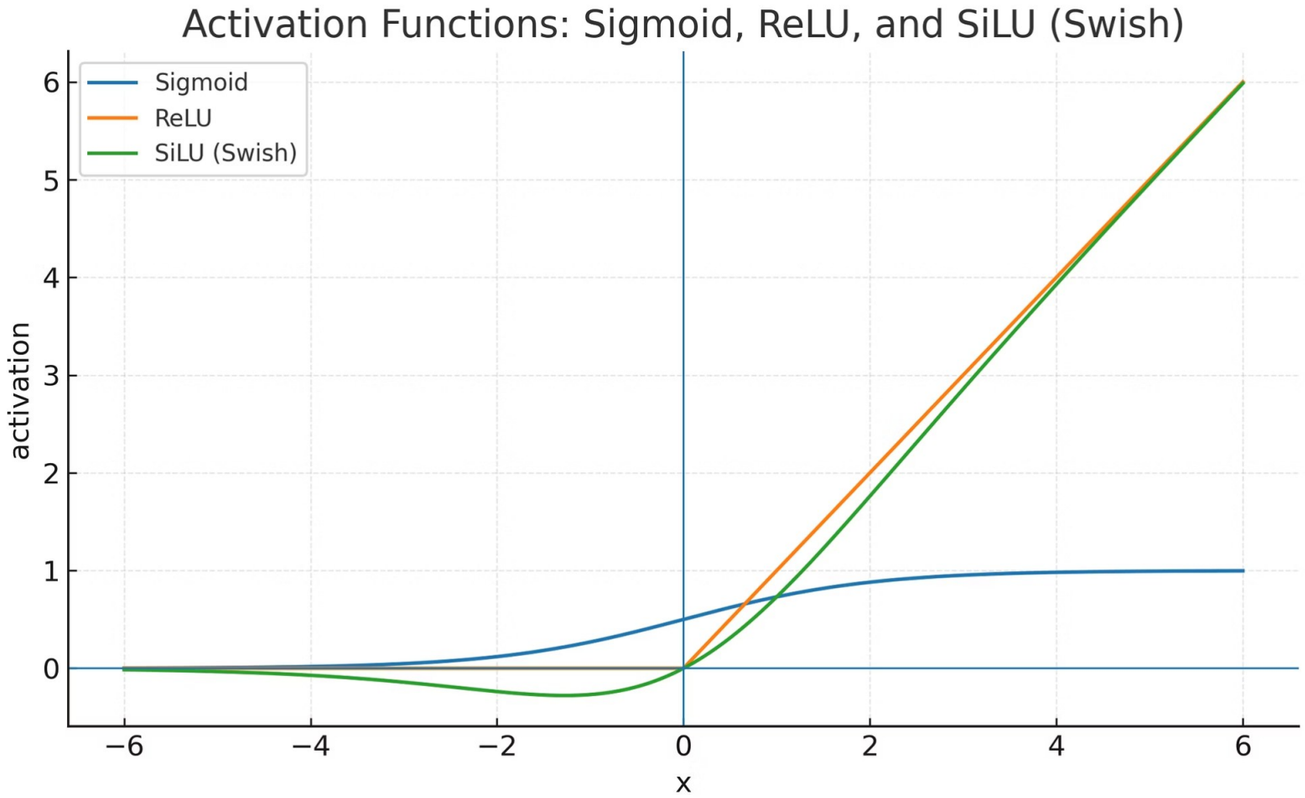
Shazeer, N. 2020. GLU Variants Improve Transformer. https://arxiv.org/abs/2002.05202

Activation Function Comparison

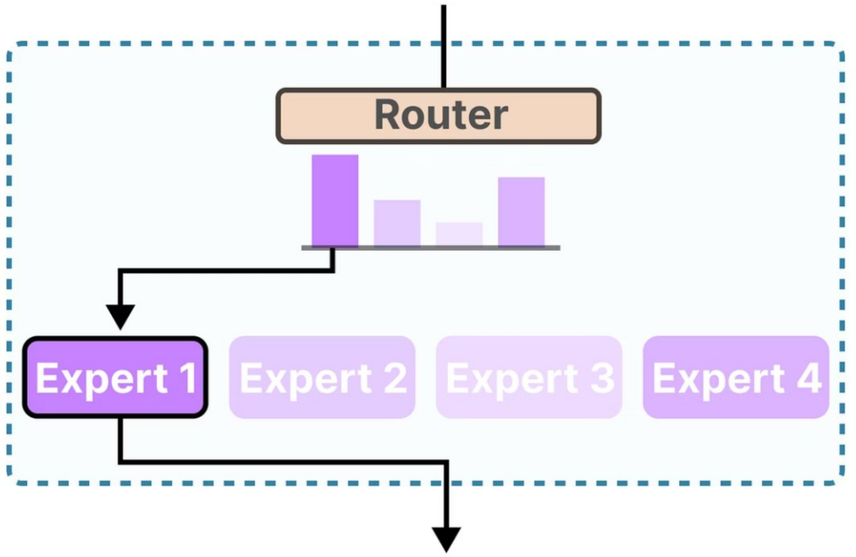
Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$

$\mathrm{ReLU}(x) = \max(0, x)$

$\mathrm{SiLU}(x) = x \cdot \frac{1}{1+e^{-x}}$



Part III: Mixture-of-Experts



Introduction to Mixture-of-Experts (MoE)

What is MoE?

A neural network architecture that combines multiple "expert" networks, each specializing in different aspects of the input space.

Key components:

- Multiple expert networks (typically FFNs)
- Router/gating mechanism
- Sparse activation (only a subset of experts processes each input)

Primary advantage: Enables parameter scaling while keeping computational costs manageable.

GShard (Lepikhin et al., 2021)

- Top-2 routing strategy
- Each token is routed to 2 out of N experts
- Pioneered MoE for large-scale language models

Switch Transformer (Fedus et al., 2021)

- Top-1 routing strategy
- Each token is routed to 1 out of N experts
- Simplified routing for better scaling

Hash Layer (Roller et al., 2021)

- Fixed routing based on hash functions
- No learned routing parameters
- More stable trainingMoE architecture enables selective activation of experts based on input tokens

Understanding Mixture-of-Experts (MoE)

In standard Transformer models, each layer contains a Feed-Forward Network (FFN). MoE replaces these FFNs with multiple "expert" networks and a routing mechanism that selects which experts to use for each token.

1

Standard MoE Architecture

In conventional MoE, each token is routed to a small subset of experts (typically 1-2 out of N). The output is computed as:

$$\mathbf{h}_t^l = \sum_{i=1}^{N} \left(g_{i,t} \, \mathrm{FFN}_i \left(\mathbf{u}_t^l \right) \right) + \mathbf{u}_t^l$$

Where gi,t is the gate value (0 or 1) determining if expert i processes token t.

2

Routing Mechanism

The router determines which experts process each token using a learned mechanism:

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{Topk}(s_{j,t} | 1 \leq j \leq N, K), \\ 0, & \text{otherwise}, \end{cases}$$

Where si,t is the token-to-expert affinity score calculated as:

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^{l^T} \mathbf{e}_i^l \right)$$

D Lepikhin et al. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In ICLR 2021.

Limitations of Conventional MoE

Knowledge Hybridity

With limited number of experts (e.g., 8 or 16), tokens assigned to a specific expert cover diverse knowledge domains.

Result: Each expert must assemble vastly different types of knowledge in its parameters, which are difficult to utilize simultaneously.

Knowledge Redundancy

Tokens assigned to different experts may require common knowledge.

Result: Multiple experts converge in acquiring shared knowledge, leading to parameter redundancy.

DeepSeekMoE Overview

40%

Computation

Compared to equivalent dense models

16B

Parameters

Total model size

2T

Tokens

Training data (English & Chinese)

40GB

GPU Memory

Can run on a single GPU

Key innovations: Fine-Grained Expert Segmentation & Shared Expert Isolation

Dai, Damai, et al. "Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models." arXiv preprint arXiv:2401.06066 (2024).https://arxiv.org/pdf/2401.06066

Fine-Grained Expert Segmentation

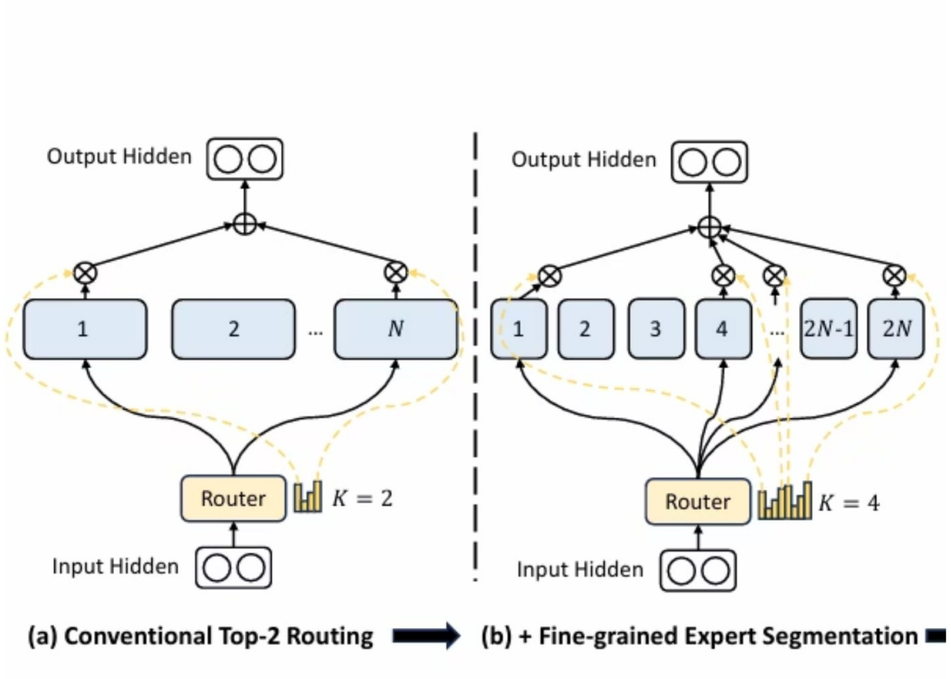
$$\mathbf{h}_t^l = \sum_{i=1}^{mN} \left(g_{i,t} \, \mathrm{FFN}_i \left(\mathbf{u}_t^l \right) \right) + \mathbf{u}_t^l$$

Where each expert is segmented into m smaller experts, and the gate values are determined by:

$$g_{i,t} = \begin{cases} s_{i,t}, & \text{if } s_{i,t} \in \mathrm{Top}k \left(\{ s_{j,t} \mid 1 \leq j \leq mN \}, mK \right), \\ 0, & \text{otherwise.} \end{cases}$$

And the token-to-expert affinity remains:

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^{l^T} \mathbf{e}_i^l \right)$$



Example: Combinatorial Flexibility

Consider a standard MoE with N=16 experts where each token activates K=2 experts:

- Possible expert combinations: $\binom{16}{2} = 120$

With DeepSeekMoE using *m=4* (splitting each expert into 4):

- Total experts: *mN = 64*
- Experts activated per token: *mK = 8*
- Possible expert combinations: $\binom{64}{8} = 4{,}426{,}165{,}368$

This dramatic increase in combinatorial flexibility enables more precise knowledge specialization.

Shared Expert Isolation

With shared expert isolation, the output of a DeepSeekMoE layer is given by:

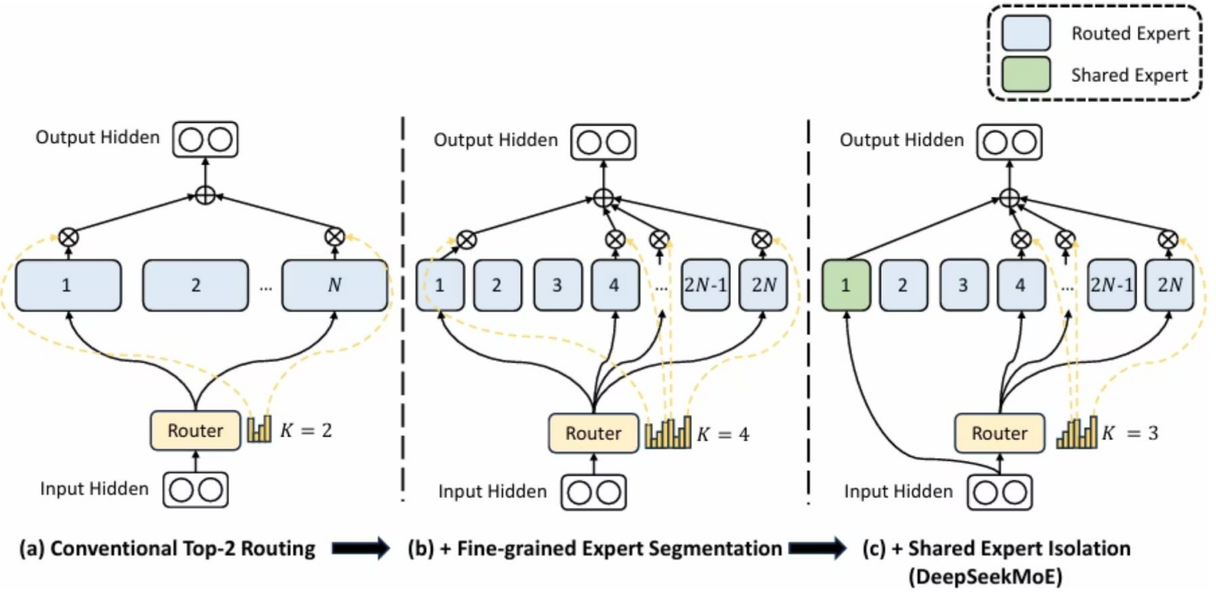
$$\mathbf{h}_t^l = \sum_{i=1}^{K_s} \mathrm{FFN}_i(\mathbf{u}_t^l) + \sum_{i=1}^{mN} \left(g_{i,t}\, \mathrm{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l$$

Where the gate values for each expert are determined by:

$$g_{i,t} = \begin{cases} s_{i,t}, & i \in \mathrm{Topk}(\{s_{j,t} \mid K_s + 1 \le j \le mN\}, mK - K_s) \\ 0, & \text{otherwise} \end{cases}$$

And the token-to-expert affinity remains:

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^{l^T} \mathbf{e}_i^l \right)$$



Part IV: TranSQL+

Serving Large Language Models with SQL on Low-Resource Hardware

[Research paper SIGMOD'26, SIGMOD'25 best demo runner-up]

Wenbo Sun (PhD candidate)

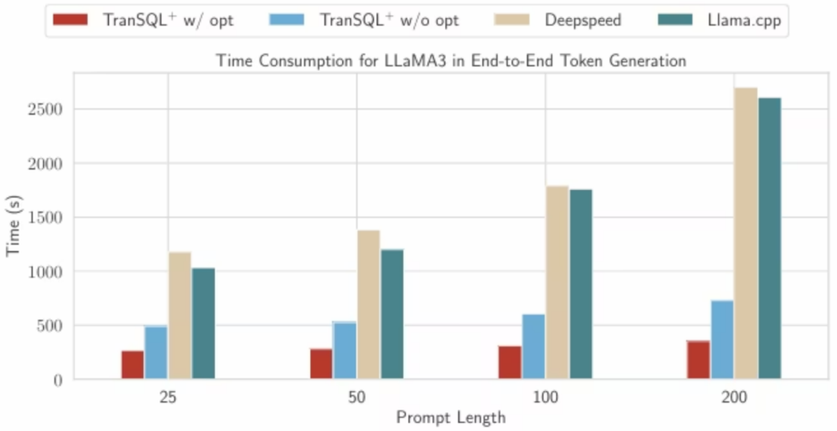


Figure 1: End-to-end time consumption. The output token length is 50. TranSQL+ shows 10× faster end-to-end (Prefill & Decoding) performance than production-ready frameworks.

Motivation: Why Run LLMs on Edge Devices?

Privacy Concerns

Integration with Local Tools

Domain-Specific Applications

Current Challenges in Edge LLM Deployment

Memory Constraints

Even smaller models exceed available memory on edge devices:

- Llama3 8B requires 16.1GB RAM
- Most laptops/smartphones have 8-16GB

Hardware Heterogeneity

Edge environments span diverse architectures:

- Personal computers (x86_64)
- Mobile/embedded (ARM v8/v9)
- IoT devices (RISC-V and others)

Existing Approaches & Limitations

- Compression methods: Distillation, quantization, pruning reduce memory but degrade accuracy
- Weight offloading: Moving weights between GPU/CPU memory, but mainly optimized for GPU systems
- Memory-mapped I/O: Frameworks like Llama.cpp support swapping from disk to memory, but introduce I/O bottlenecks
- Porting complexity: Supporting new operators and hardware requires re-engineering and specialized skills

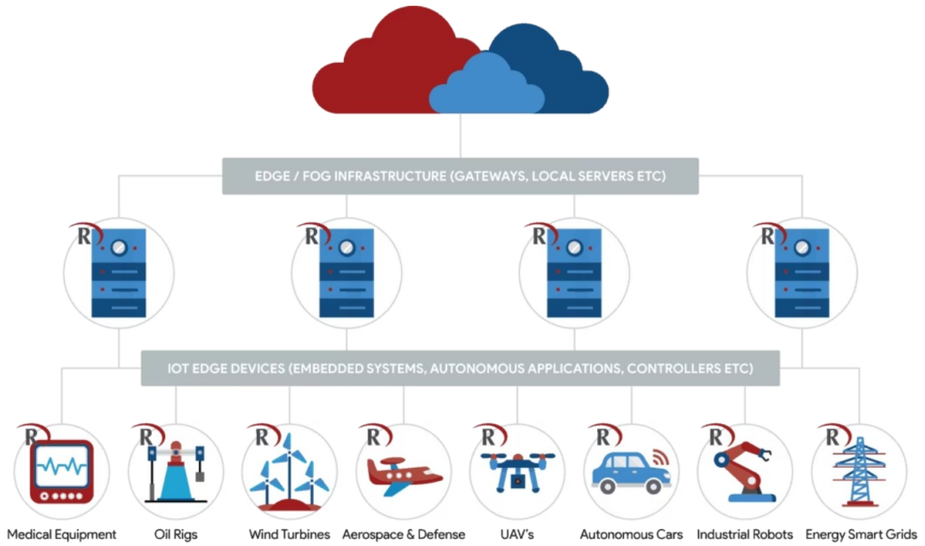
Our Proposal: Leveraging Relational Databases

Relational databases offer several advantages for LLM inference:

- Ubiquitous presence (e.g., SQLite embedded in billions of devices)
- Mature out-of-core execution capabilities
- Efficient cache management
- Vectorized execution for performance
- SQL's inherent portability across diverse platforms

Databases align well with interactive workloads like LLM inference under tight memory budgets.

Databases can efficiently manage model parameters that exceed RAM and accelerate linear-algebra kernels via batch-oriented, vectorized execution.



Key Idea & Contributions

Core Idea: Translate the LLM computation graph into pure SQL, execute in a relational DBd

C1

Template-based SQL Generator

First template-based SQL code generator to serve modern LLMs inside DB engines.

C2

Optimization Techniques

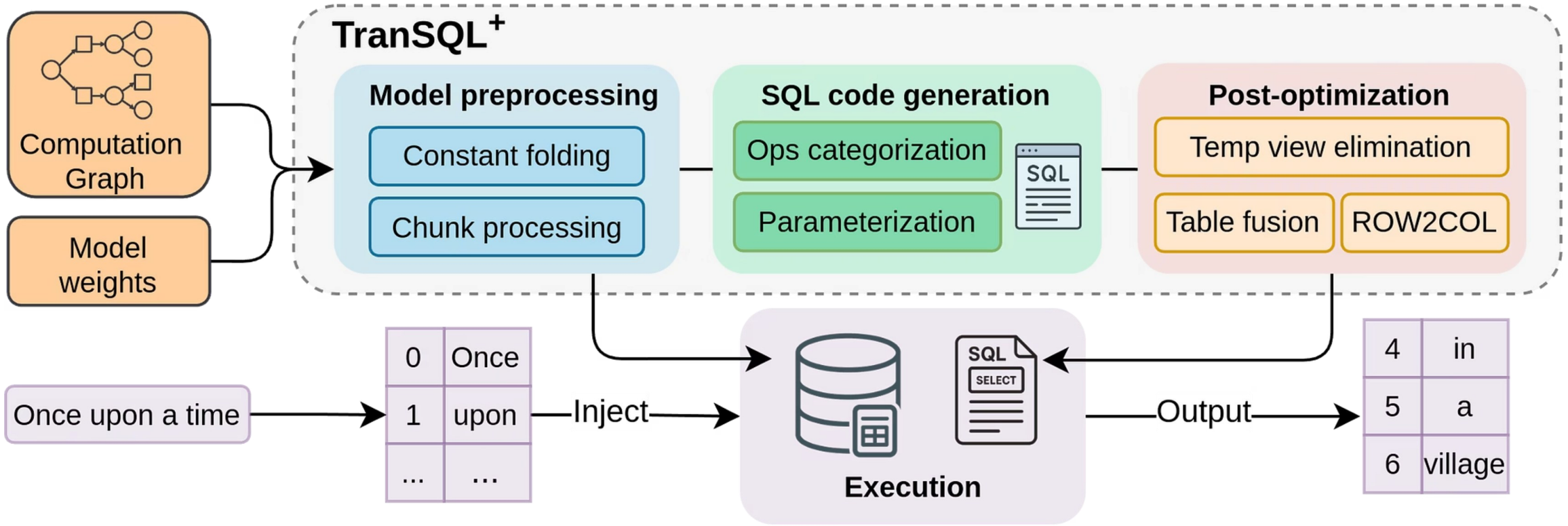
ROW2COL pivot + table fusion reduce intermediate cardinality and boost vectorized execution.

C3

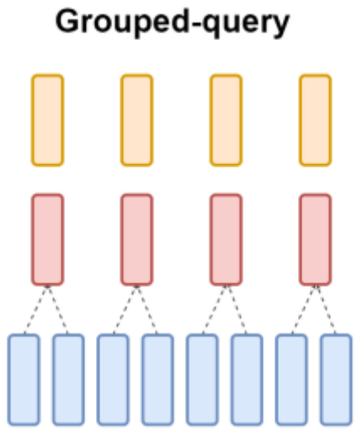
Performance Gains

On CPU-only, low-memory hardware: up to 20× lower prefill latency, up to 4× faster decoding vs DeepSpeed & Llama.cpp.

An overview of the workflow of TranSQL+



Recap



LLaMA-3 8B
Model dimension d_k= 4096
Number of attention heads = **32**
So, per head dimension = **4096 ÷ 32 = 128**

For **keys (K)**:

$$W_K = [W_{K,1} W_{K,2} \ \dots W_{K,32}]$$

Where each $W_{K,h} \in \mathcal{R}^{4096x128}$

For **values (V)**:

$$W_V = [W_{V,1} W_{V,2} \ \dots W_{V,32}]$$

Where each $W_{V,h} \in \mathcal{R}^{4096x128}$

Model Preprocessing

1. Chunk-Based Representation

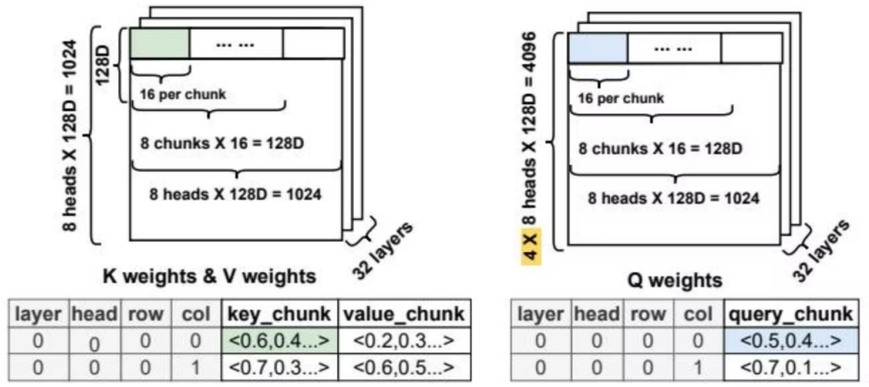
To execute neural operators via SQL, model weight matrices are converted into relational tables using a chunked format:

- Matrices are split into smaller blocks, indexed by row and column
- Each row is split into chunks of fixed size
- For a matrix $W \in \mathbb{R}^{m \times n}$, each row is represented as:

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

In the database table, each row corresponds to:

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



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

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

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

This chunking strategy provides flexibility by allowing trade-offs between the number of rows and the workload per thread when performing vector operations.

Model Preprocessing

2 Constant Folding

Precompute constant tensors and any downstream linear or scalar operations

Benefit: avoid redundant computation at inference time

Example: attention scaling factor $\frac{1}{\sqrt{d_k}}$

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

Template-based SQL Code Generation

TranSQL+ translates LLM inference computations into executable SQL using a template-based approach:

- Each neural operator maps to a predefined SQL template
- Templates are instantiated with model-specific parameters
  - For instance, attention-head group size (GQA) and token/embedding dimensionality
- Operators are categorized by type and parameterized based on their dimensions
  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)
  v. Normalization-related operations (e.G., Rmsnorm, softmax)

Template-based SQL Code Generation

A neural operator acting on p operands is represented as:

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

Where:

- $\mathcal{F}$ denotes the set of free dimensions retained in the output (grouping or projection columns)
- $\mathcal{S}$ encodes the mappings of shared dimensions between operands (translated to SQL join keys)
- $\mathcal{G}$ represents the dimensions used in GROUP BY clauses
- attr indicates operator-specific attributes that guide the projection logic

Example: Matrix Multiplication in SQL

For matrix multiplication X = AB, where A ∈ $\mathbb{R}^{mxr}$ and B ∈ $\mathbb{R}^{rxn}$, each element is calculated as:

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

Matrix multiplication *X = AB* is parameterized as:

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

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

Where:

- A and B are input matrices
- m_A, n_B are group-by dimensions (output row/column indices)
- $r_A,\ r_B$ are shared dimensions for summation

Example: Matrix Multiplication in SQL

Within the chunked table structure, matrix multiplication is implemented using JOIN and GROUP BY operations in below pattern:

```
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;
```

```
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
```

This SQL-based matrix multiplication serves as a foundation for implementing other operations directly in the relational engine.

Templates for Primary Operator Categories

Operator Category	Operator	Template	SQL Query
Matrix multiplication	AB	$R_3 \leftarrow \Pi_{\mathcal{G},\mathcal{S},\mathrm{DOT}(\mathbf{c}_1,\mathbf{c}_2)}(R_1 \bowtie_{\mathcal{S}} R_2)$ $\Pi_{\mathcal{G},\mathrm{SUM}(\mathbf{c}_3)}(\gamma_{\mathcal{G}}(R_3))$	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)	$\Pi_{\mathcal{F},f(\mathbf{c}_1)}(R_1)$	SELECT A.m, A.n, 1/(1+exp(-A.chunk)) FROM A
Element-wise arithmetic	$A+B$	$\Pi_{\mathcal{S},f(\mathbf{c}_1,\mathbf{c}_2)}(R_1 \bowtie_{\mathcal{S}} R_2)$	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()	$\Pi_{\mathcal{G},f(\mathcal{S}),\mathbf{c}_1}(R_1)$	SELECT A.m*M+A.n, A.chunk FROM A
Normalization	Softmax(A)	$R2 \leftarrow \Pi_{\mathcal{F},\mathcal{G},\mathrm{agg}(f(\mathbf{c}_1))}(R_1)$ $R3 \leftarrow \Pi_{\mathcal{G},\mathrm{agg}(\mathbf{c}_2)}\gamma_{\mathcal{G}}(R_2)$ $\Pi_{\mathcal{F},\mathcal{G},g(\mathbf{c}_2,\mathbf{c}_3)}(R_2 \bowtie_{\mathcal{S}} R_3)$	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

These templates cover the core neural operators used in transformer-based LLMs like Llama3 and DeepSeek MoE.

Why templates?

- **Extensibility**: easy to add new operators, simply add templates
- Examples: transformer components, convolution kernels

Post-optimization I: Temporary View Elimination

Problem: Initial SQL generation produces many subqueries and temporary views, causing significant I/O overhead.

Solution: Leverage Common Table Expressions (CTEs) to merge multiple subqueries into a single, unified SQL statement, reducing I/O from repeated materialization and scans

1. Step 1: Topological sort the computation graph
2. Step 2: Visit each node and merged them into CTE block.

   For critical nodes: the currently accumulated nodes are materialized into a table

   For other nodes: absorb nodes into the ongoing CTE expression
3. Step 3: Combine all SQL statements into a final script

☆ Critical nodes: (i) is consumed by multiple downstream operators or (ii) is subsequently modified by one or more operators

- Key and Value Vectors of Attention Heads
- Embeddings for Residual Connections

Post-optimization II: Table Fusion

Problem: In LLM inference, query, key, and value vectors are computed independently but share the same input, resulting in redundant scans and multiple temporary tables.

Solution: Concatenate the projection weights into a single extended matrix:

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

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

Benefits: Merging Q/K/V weight tables allows scanning weights once and computing all three matrix multiplications in a single SQL query, reducing cache flushes and repeated scans while exposing finer-grained parallelism.

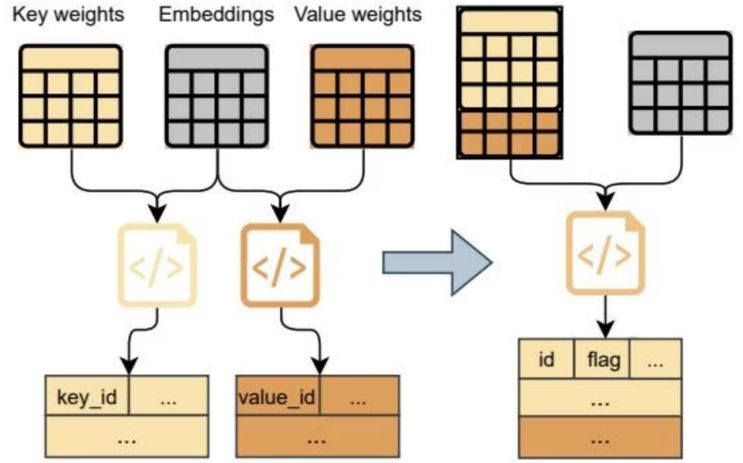
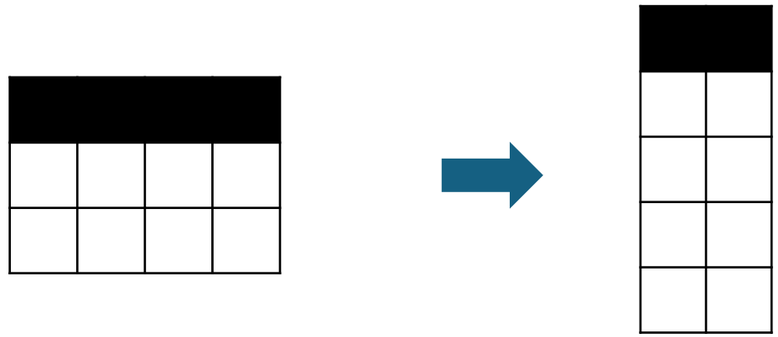


Figure 5: Merge computation of key and value vectors into a single CTE.

Post-optimization III: ROW2COL Pivoting

Problem: Join-based alignment in relational engines inflates intermediate results, forces key materialization, and limits intra-operator parallelism.

Solution: ROW2COL pivots chunk rows into columns, reducing join cardinality and exposing contiguous, independent column slices for parallel processing.



Benefits: (1) Reduces source rows and intermediate join cardinality, (2) Exploits vectorized engines by treating columns as contiguous arrays, (3) Exposes column-aligned chunks for parallel processing.

Experimental Setup

Models Tested

- Llama3-8B (dense, GQA, SwiGLU)
- DeepSeek-MoE (6 experts activated)

Hardware Configuration

- 4 CPU cores
- 16 GB RAM (also tested down to 8 GB)

Database Backends

- DuckDB
- ClickHouse

Baselines & Metrics

- DeepSpeed Inference, Llama.cpp (CPU out-of-core)
- Prefill latency (to first token)
- Per-token decoding latency

All experiments conducted in CPU-only, memory-constrained scenarios to simulate edge device deployment.

Prefill and decoding latency

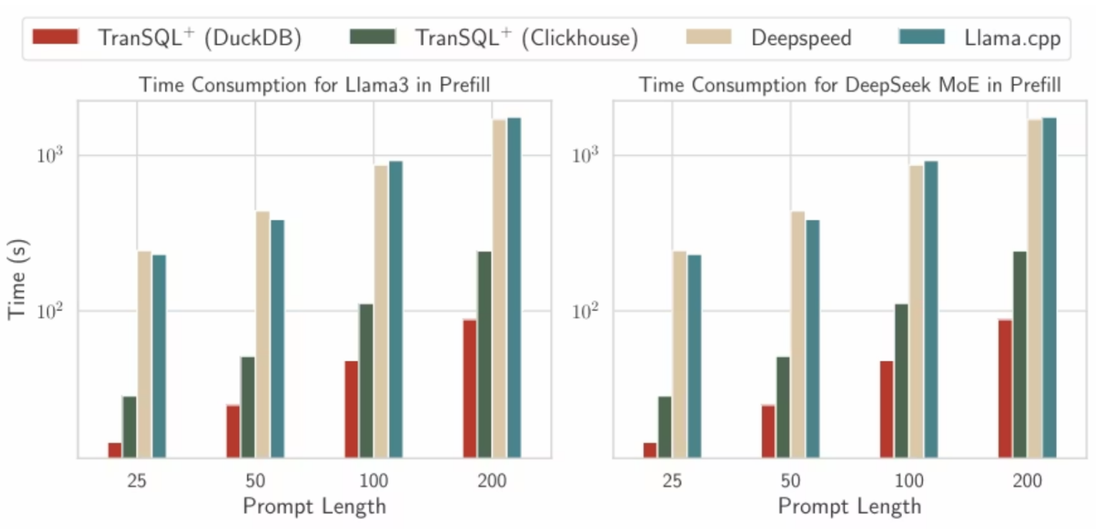


Figure 6: Prefilllatency with varying prompt length. TranSQL+ gains up to 20x speedups compared to Deep speed and Llama.cpp.

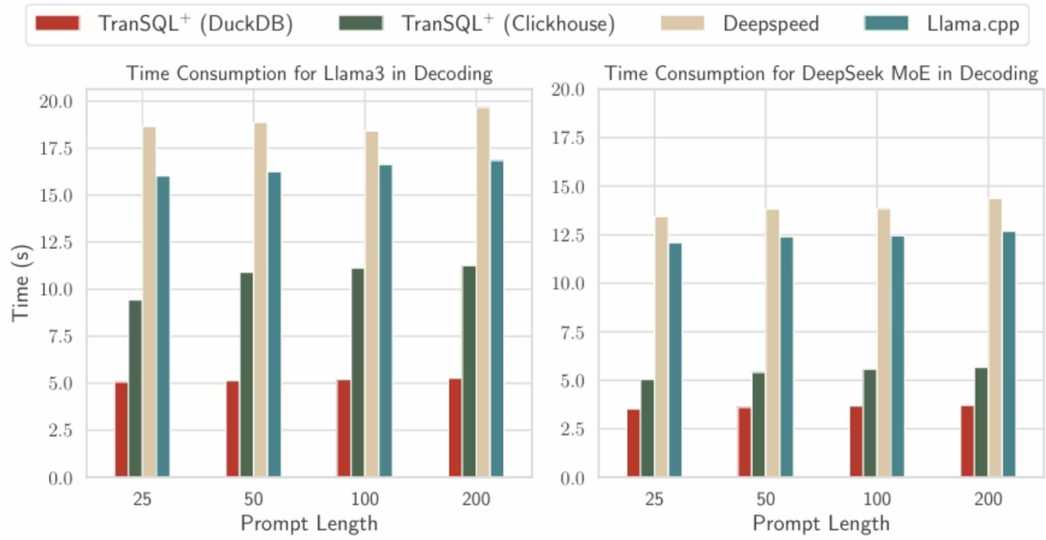


Figure7: Decoding latency with varying prompt length.TranSQL+gains up to 4x speedups compared to Deepspeed and Llama.cpp.

TranSQL+ outperforms existing inference frameworks on resource-constrained hardware.

Impact of Post-optimizations

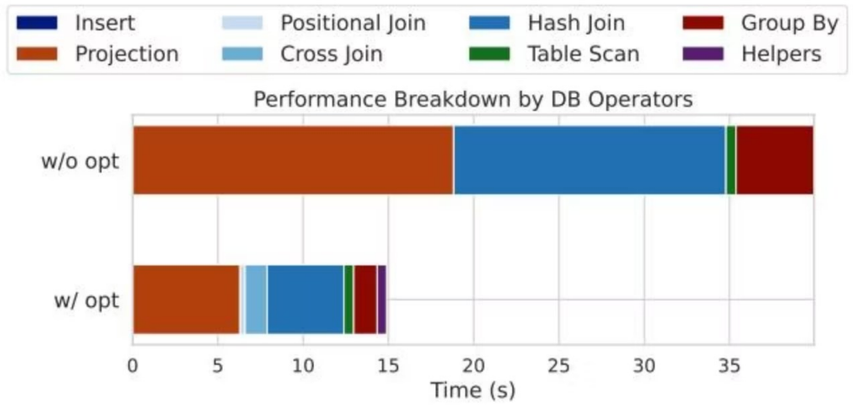


Figure 9: Join time drops by over 60% because the optimizations reduce intermediate cardinality.

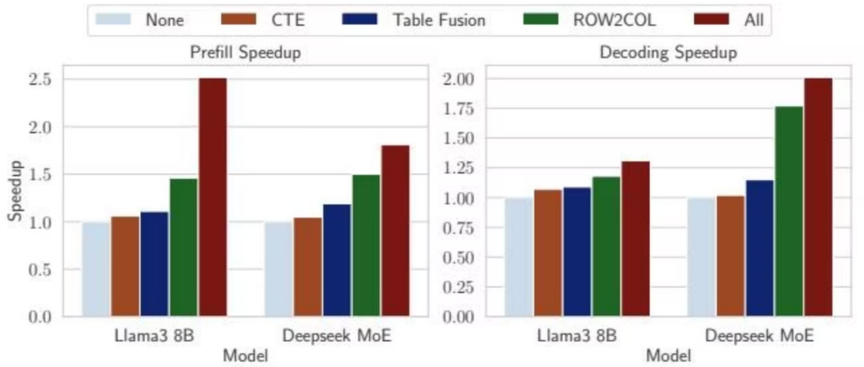


Figure 10: Ablation study for three post optimizations.

- The post-optimization strategy reduces total execution time by 64.8% compared to unoptimized SQL queries.
- Among the three optimizations, ROW2COL has the most dramatic effect by exploiting columnar storage and SIMD-parallel operations.

Influence of Memory Capacity

Figure 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.

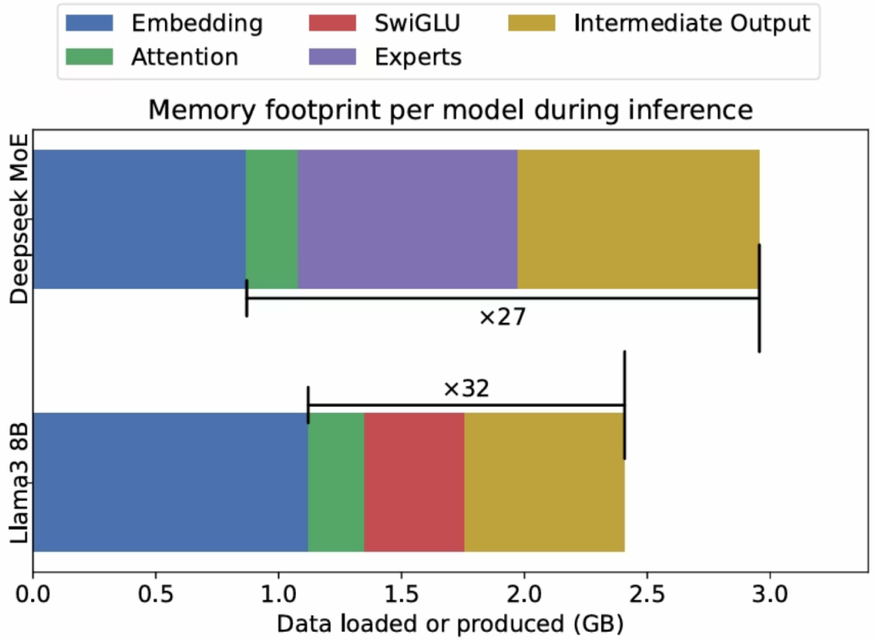


Figure 13: Prefill memory usage with 50-token input.

- Memory capacity is critical for efficient LLM inference on low-resource hardware.
- Adequate memory reduces latency in both prefill and decoding by minimizing data movement.

Scope, Generality & Limitations

Portability

Standard SQL + columnar/vectorized engines; applies to dense & MoE; CNNs via im2col mapping.

Performance Context

Not a GPU replacement: single A100 still much faster; goal is practical on CPU, with little RAM.

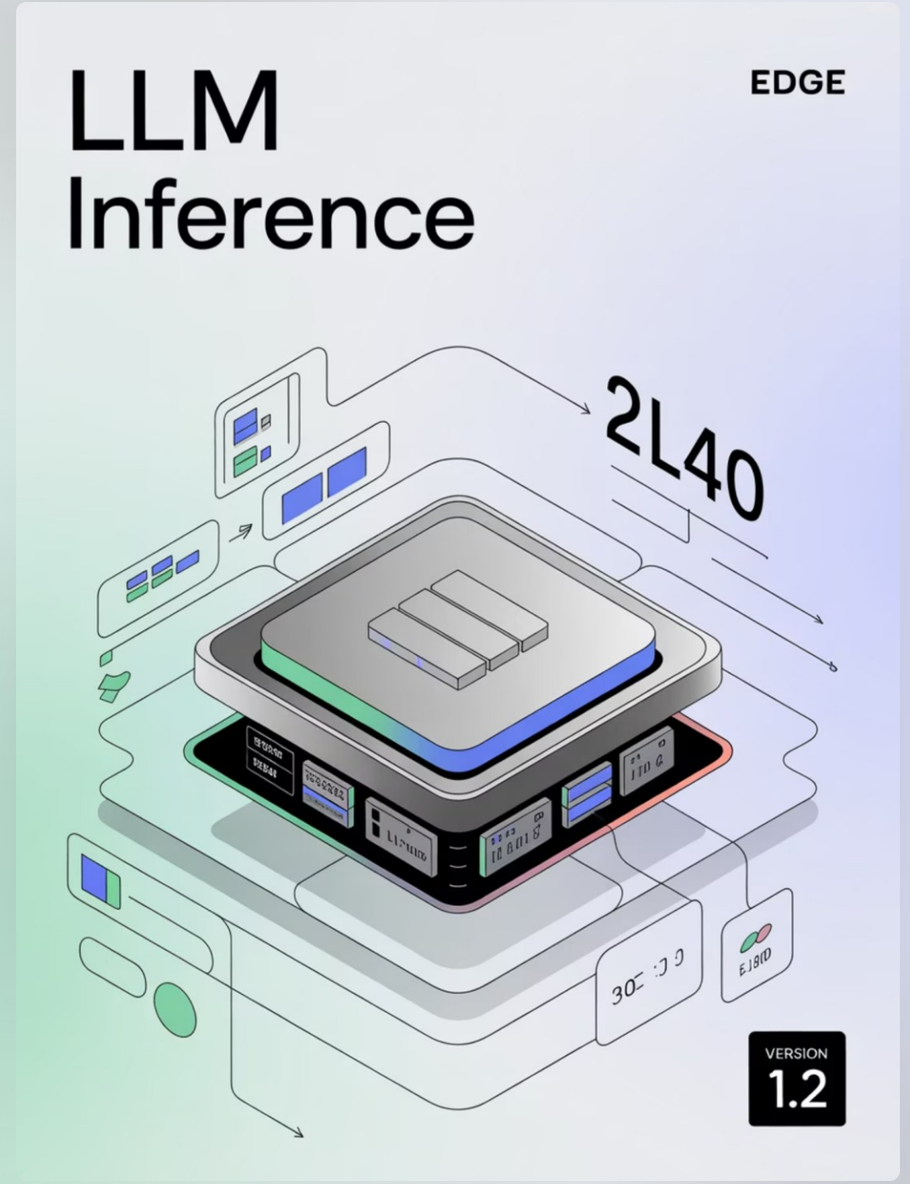
Complementary Techniques

Quantization/distillation not used; can be layered on later for additional gains.

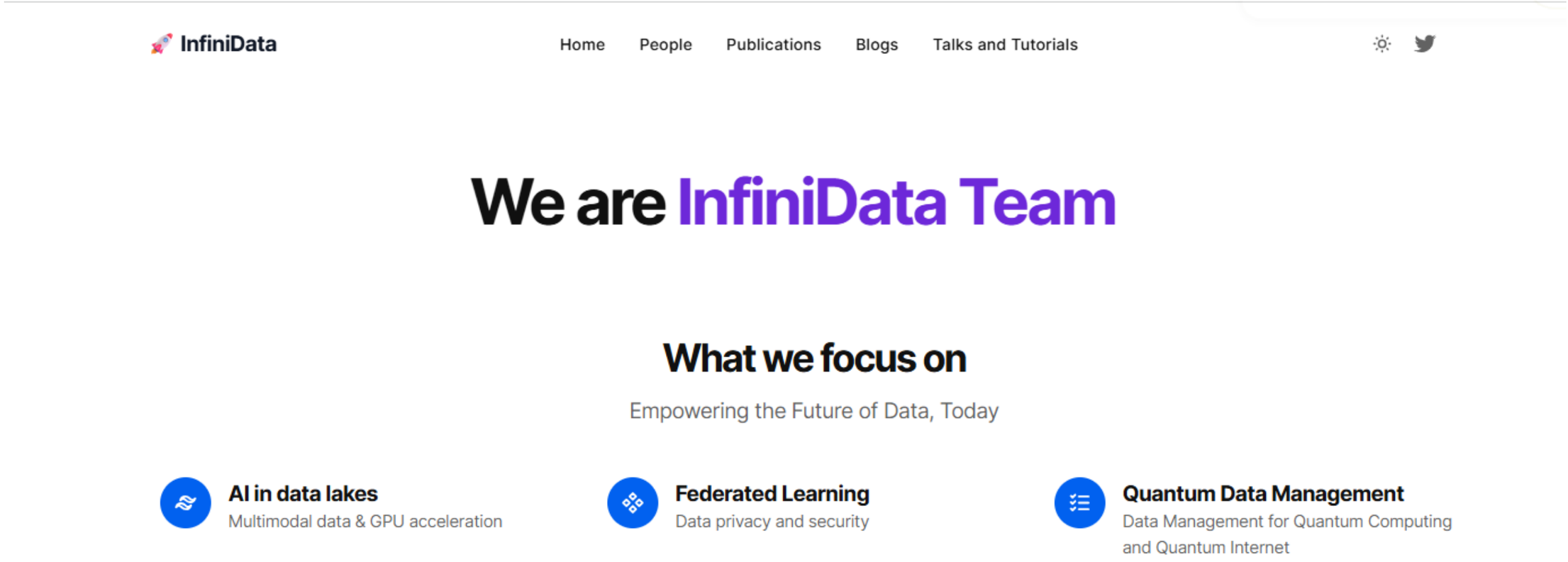
Engine Considerations

ROW2COL/positional join support varies; cost-modeling left for future work.

- Use when GPUs aren't available and memory is tight.



Slides will be on our team webpage



Scalability problem in simulating quantum computation

- We can represent an **n**-qubit quantum state as a vector of size 2^n

$$|\psi\rangle = \alpha_{0\dots0}|0\dots0\rangle + \alpha_{1\dots1}|1\dots1\rangle = \begin{bmatrix} \alpha_{0\dots0} \\ \alpha_{0\dots1} \\ \\ \cdots \\ \\ \alpha_{1\dots0} \\ \alpha_{1\dots1} \end{bmatrix} \Bigg\} \text{ Vector size: } 2^n$$

- Reaching the memory limits of today's supercomputers

nature physics ARTICLES
https://doi.org/10.1038/s41567-018-0124-x

Characterizing quantum supremacy in near-term devices

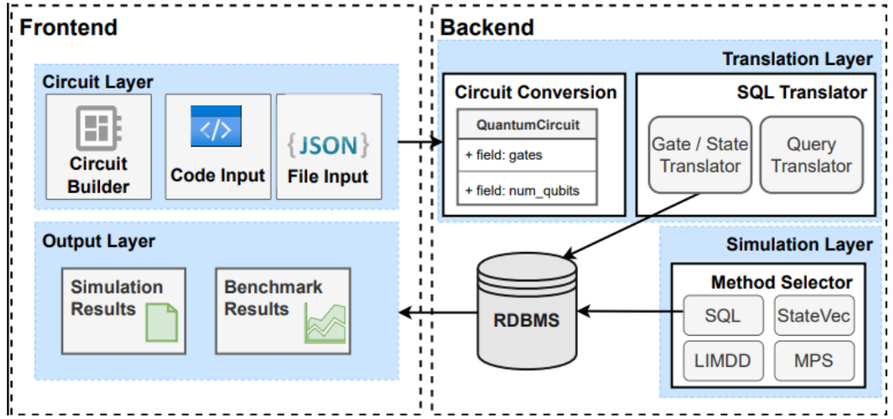
Sergio Boixo[1*], Sergei V. Isakov[2], Vadim N. Smelyanskiy[1], Ryan Babbush[1], Nan Ding[1], Zhang Jiang[3,4], Michael J. Bremner[5], John M. Martinis[6,7] and Hartmut Neven[1]

2.25 petabytes for **48** qubits (single precision)

Efficient tensor computation: database to the rescue

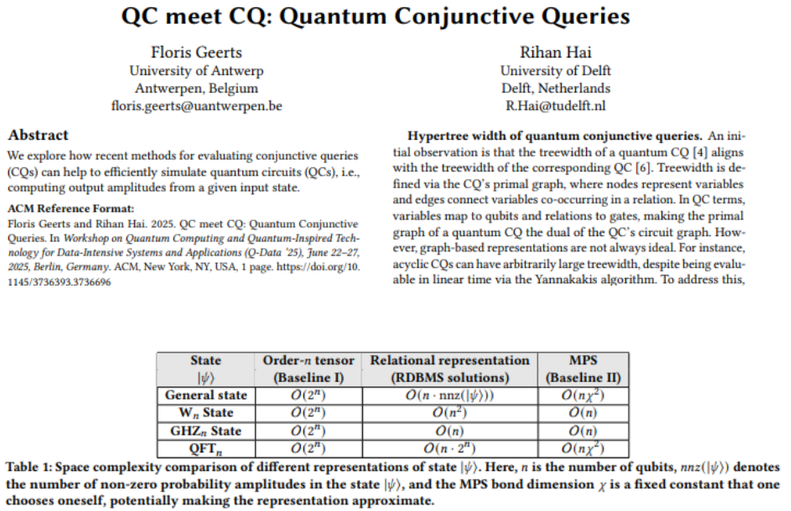
Q1: Push the simulation workload to DBMSs?

System



Littau, Tim, and Rihan Hai. "Qymera: Simulating Quantum Circuits using RDBMS." SIGMOD. 2025.

Theory



QC meet CQ: Quantum Conjunctive Queries

Floris Geerts
University of Antwerp
Antwerpen, Belgium
floris.geerts@uantwerpen.be

Rihan Hai
University of Delft
Delft, Netherlands
R.Hai@tudelft.nl

Abstract

We explore how recent methods for evaluating conjunctive queries (CQs) can help to efficiently simulate quantum circuits (QCs), i.e., computing output amplitudes from a given input state.

ACM Reference Format:
Floris Geerts and Rihan Hai. 2025. QC meet CQ: Quantum Conjunctive Queries. In *Workshop on Quantum Computing and Quantum-Inspired Technology for Data-Intensive Systems and Applications (Q-Data '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3736393.3736696

Hypertree width of quantum conjunctive queries. An initial observation is that the treewidth of a quantum CQ [4] aligns with the treewidth of the corresponding QC [6]. Treewidth is defined via the CQ's primal graph, where nodes represent variables and edges connect variables co-occurring in a relation. In QC terms, variables map to qubits and relations to gates, making the primal graph of a quantum CQ the dual of the QC's circuit graph. However, graph-based representations are not always ideal. For instance, acyclic CQs can have arbitrarily large treewidth, despite being evaluable in linear time via the Yannakakis algorithm. To address this,

State $\lvert\psi\rangle$	Order-n tensor (Baseline I)	Relational representation (RDBMS solutions)	MPS (Baseline II)
General state	$O(2^n)$	$O(n \cdot \mathrm{nnz}(\lvert\psi\rangle))$	$O(n\chi^2)$
$\mathbf{W}_n$ State	$O(2^n)$	$O(n^2)$	$O(n)$
$\mathbf{GHZ}_n$ State	$O(2^n)$	$O(n)$	$O(n)$
$\mathbf{QFT}_n$	$O(2^n)$	$O(n \cdot 2^n)$	$O(n\chi^2)$

Table 1: Space complexity comparison of different representations of state $\lvert\psi\rangle$. Here, n is the number of qubits, $nnz(\lvert\psi\rangle)$ denotes the number of non-zero probability amplitudes in the state $\lvert\psi\rangle$, and the MPS bond dimension χ is a fixed constant that one chooses oneself, potentially making the representation approximate.

Geerts, Floris, and Rihan Hai. "QC meet CQ: Quantum Conjunctive Queries." *Proceedings of the 2nd Workshop on Quantum Computing and Quantum-Inspired Technology for Data-Intensive Systems and Applications*. 2025.

Hai, Rihan, et al. "Quantum Data Management in the NISQ Era: Extended Version." arXiv preprint arXiv:2409.14111 (2024).