

AI Query Compilation for Unified and Optimized Execution

Yeounoh Chung*
Google Cloud
Sunnyvale, CA, USA
yeounoh@google.com

Helena Caminal
Google Cloud
Sunnyvale, CA, USA
hcaminal@google.com

Fatma Özcan
Google Cloud
Sunnyvale, CA, USA
fozcan@google.com

ABSTRACT

In this vision paper, we propose a novel architectural paradigm for accelerated AI query execution via a unified compiled execution strategy. By compiling the hybrid *AI Query* as a whole—integrating both standard SQL relational constructs and LLM inference layers into a single, unified tensor compute graph—we completely alleviate PCIe data movement bottlenecks across execution boundaries and enable global compiler optimizations and efficient automatic sharding. We demonstrate the viability of this unified execution paradigm on select and extended AI queries on *SemBench* Reviews and Movies datasets, achieving up to 5.3x latency speedup and 9.8x throughput speedup on TPUs, and outline a research roadmap of open technical challenges to realize this vision.

VLDB Workshop Reference Format:

Yeounoh Chung, Helena Caminal, and Fatma Özcan. AI Query Compilation for Unified and Optimized Execution. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

PVLDB Artifact Availability:

N/A

1 INTRODUCTION

Relational query processing through hardware accelerators, such as GPUs and TPUs, has been an active area of research [4, 11–13, 19, 22, 24–26, 29]. However, when it comes to AI queries or SQL queries augmented with AI semantic operators (e.g., using an LLM for semantic classification or extraction) [5, 20, 21, 23], current systems typically adopt a split execution model. Traditional SQL operations are executed on the CPU or a separate database engine, while LLM inference is offloaded to a dedicated ML serving system, often running on a remote machine or a separate accelerator.

Unfortunately, this dichotomy creates a two-fold penalty that limits the performance of modern AI queries. First, it imposes substantial data movement overheads, as intermediate results must be transported across the Host-To-Device (H2D) boundaries between the database engine’s system DRAM and the inference engine’s high-bandwidth memory (HBM) or network-attached remote computing resources such as Google Vertex AI [10]. This fragmentation exacerbates the classic *Memory Wall* [27], where the performance of the system is dictated not by raw compute capability, but by the bandwidth and latency of data movement [8]. In the AI era, this

has evolved into a *Data Movement Wall* [14], where the cost of marshaling data between isolated execution environments outweighs the gains of specialized hardware acceleration. Second, these rigid execution boundaries can also act as an optimization huddle, precluding any form of global AI query planning. Because the SQL optimizer remains blind to the downstream neural operations, while the ML compiler also remains unaware of the upstream relational context, potential cross-boundary optimizations (e.g., SQL and AI operator fusion) are overlooked.

In this paper, we propose a paradigm shift toward a unified execution strategy for AI queries. Our key insight is that since modern LLM inference models are fundamentally executed as tensor compute graphs, we can bridge the gap by lowering standard relational database operations (e.g., scans, filters and aggregations) into that same tensor compute graphs. By unifying relational constructs and neural layers into a consolidated executable, we can leverage the full power of modern ML compiler infra, such as JAX [7] or MLIR [17]. This allows the entire hybrid flow to be compiled into a single accelerator executable, optimized holistically for specialized hardware backends, like TPUs, GPUs or other specialized hardware backends for MLIR.

This unified approach provides more than just the data movement latency elimination; it fundamentally removes the optimization barriers due to the fragmented execution environments, unlocking new opportunities for global cross-boundary optimizations for AI queries or any analytics workloads that implement both traditional SQL and AI semantic operators. A major global optimization enabled by this unified program is unified, compiler-driven multi-device sharding. In traditional split AI Query Engine architectures, developers must manually write, calibrate, and synchronize two completely independent parallelization strategies: one for the CPU database engine (e.g., dynamic core-level partitioning) and another for the TPU/GPU machine learning engine (e.g., tensor-parallel or pipeline-parallel model sharding). This requires complex dynamic load balancing across the CPU-TPU boundary. By contrast, because our compiler translates both relational database scans and LLM inferences into a single unified tensor program, the compiler can globally shard both the relational data arrays and the neural computations concurrently and automatically via XLA’s GSPMD/PMAP. This completely eliminates the need to write separate parallelization logic for the CPU and TPU engines, allowing the compiler to dynamically and optimally partition the entire hybrid query end-to-end across hardware resources.

2 RELATED WORKS

Our work is at the intersection of relational database hardware acceleration and LLM-based AI Query engine optimizations.

Relational Algebra on Accelerators: accelerating SQL query processing on specialized hardware accelerators (e.g., GPUs and

*Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.

TPUs) has been an active area of research. Early works in tensor query processing mapped standard relational algebra operators (e.g., selections, joins and projections) to GPU tensor kernels [11, 13]. Recent research projects explore leveraging accelerator hardware for graph databases [18] and other broader ML workflows [2].

Our work is orthogonal to these relational query acceleration works. While they focus entirely on enabling highly optimized relational query processing on hardware accelerators, our work specifically targets AI queries or hybrid queries (SQL + LLMs). Our contribution is showing how to unify relational algebra and deep neural layers into the same compiled tensor program, completely consolidating the accelerator execution boundaries and their cross-boundary data movements. The unified compiled execution via JAX/XLA also enables auto-sharding, which otherwise requires separate parallelization and optimizations on CPU database engines and the TPU/GPU ML serving frameworks.

AI Query Systems & Serving Boundaries: as LLMs have grown in popularity, several contemporary data analytics systems have integrated LLM-based semantic operators into relational query engines for structured and unstructured data analytics [5, 20, 23]. These systems universally adopt a split execution model: traditional SQL or other data processing pipelines run eagerly on the CPU database host, whereas semantic operators (LLM inference) is offloaded to a local or remote model serving engine (e.g., vLLM [15]) running on GPUs/TPUs. While recent industry architectures envision unified, AI-powered agentic data clouds where AI agents and database operators run seamlessly on consolidated enterprise platforms [6], contemporary physical implementations still strictly separate relational engines from LLM serving boundaries.

3 THE CASE FOR UNIFIED AI QUERY EXECUTION

Our system achieves accelerated AI Query execution via a compiled unified execution path: compiling the AI Query as a whole, integrating both standard SQL relational constructs and the deep learning neural layers (e.g., LLM inference for semantic operators) into a single, unified compute graph. Contemporary AI Query engines operate as a fragmented mixture execution model, split by fixed hardware boundaries. Standard relational operations run eagerly on CPU database engines, whereas LLM inferences are offloaded to separate ML serving frameworks on accelerators (GPUs/TPUs). This split-boundary execution incurs severe data movement overheads and prevents global compiler optimizations. We translate both relational operators and LLM inference into a unified JAX program, compiled into a single global TPU/GPU executable via XLA JIT [7]. This consolidated approach is illustrated in Figure 3.

3.1 Translation and Mapping Mechanics

Each component of a hybrid SQL query maps to SQL AST constructs, which our compiler translates directly to standard JAX functions/primitives. Table 1 describes the structural mapping mechanics from SQL clauses to their compiled JAX implementations.

As a running example, let us consider a hybrid AI Query that filters rows using a semantic predicate evaluated by an LLM: `SELECT SUM(values_a) FROM bench_table WHERE AI_IF(text_col, 'Is this review positive?') = 1`.

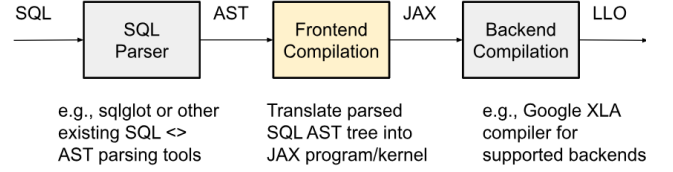


Figure 1: High-level view of the unified AI Query compilation flow.

To compile this AI Query with *AI_IF* semantic classification AI operator, the system first parses the SQL text into an Abstract Syntax Tree (AST). The AST tree is subsequently mapped and compiled into a single JAX program (Figure 2 per our running example).

```

1 @jax.jit
2 def classification_query_kernel(params, input_ids,
3   attn_mask, values, yes_id, no_id):
4   """
5   SELECT SUM(values) WHERE Model_Predicts(
6     Next-Token) == 'Yes'
7   """
8   # Logits shape: (Batch_Size, Sequence_Length,
9     Vocab_Size)
10  outputs = model(input_ids=input_ids,
11    attention_mask=attn_mask, params=params)
12
13  # Slice[:, -1, :] to get the predictions at
14    the very end of the sequence.
15  last_token_logits = outputs.logits[:, -1, :]
16
17  # Look up the specific columns for Yes and No
18  score_yes = last_token_logits[:, yes_id]
19  score_no = last_token_logits[:, no_id]
20  filter = score_yes > score_no # Filter
21
22  # Aggregation
23  return jnp.where(filter, values, 0.0).sum()
  
```

Figure 2: Actual generated JAX program containing the vectorized execution trace for the example AI Query. This is compiled via *@jax.jit* using XLA for optimized hardware execution.

3.2 Frontend Code Compilation

Our system performs frontend compilation as to generate JAX program/kernel for XLA backend compilation. The frontend code generator, if you will, reads the parsed SQL AST and traverses it to extract relational predicates, UDF (AI operators) names, and table definitions. Based on the translation rules in Table 1, the compiler retrieves matching template files written in JAX/Python from the UDF (AI operators) registry and inserts them into the global JAX program/function template. Figure 3 illustrates the code generation flow.

Note that JAX provides high-performance control flow operator, such as *jax.lax.scan*, which expresses the loop structure (e.g., for,

SQL Clause / Operator	AST Node (sqlglot)	Physical Translation	Compiled JAX Implementation
FROM table	exp.From	Input Array Loading	Standard JAX input arrays packed into host sharded memory (input_ids, attn_mask, values_a).
WHERE values_a < 0.2	exp.Where	Boolean Masking	Vectorized JAX boolean array mask generated dynamically: values_a < 0.2 (evaluates to True / False array).
AI_IF(text_col) = 1	exp.Func (WHERE)	LLM Classification	High-level UDF calling Flax causal LM forward pass. Logits for “Yes”/“No” are compared on-chip to yield boolean predicates.
AI_TRANSFORM(text_col)	exp.Func (SELECT)	LLM Generation	Flax causal LM forward pass with generative decoding, output sharded and masked via jnp.where and pad_id.
SELECT SUM(values_a)	exp.Agg	Masked Reduction	Vectorized summation following the vectorized JAX boolean array masking: jnp.where(final_predicate, values_a, 0.0).sum().

Table 1: SQL to JAX translation mapping examples

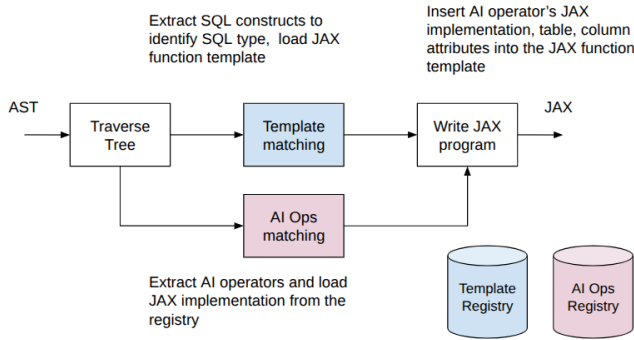


Figure 3: Frontend compilation flow from SQL AST to unified JAX program.

while) without unrolling into a huge static program. This is crucial for memory management, as the generated scan loop streams batches of data through the LLM forward pass directly on the TPU, maintaining the intermediate states in TPU registers.

3.3 H2D Boundary Consolidation

By compiling both standard relational operations (e.g., *WHERE* filters and *SUM* aggregations) and LLM UDFs into the same compiled trace, we achieve host to accelerator device (H2D) boundary consolidation. In the baseline, mixture execution model, data is moved back and forth between the CPU database engine and the TPU model serving engine:

- **H2D (Host-to-Device)** copy of the filtered database rows into TPU HBM memory.
- **LLM Inference** model inference for semantic operators executed on the TPU
- **D2H (Device-to-Host)** copy of the large output logits back to CPU Host RAM
- **Host Processing** data processing and aggregation, like *SUM*, executed on CPU.

The D2H copy of output logits can be expensive. For a typical batch size of 128 and sequence length of 32, and Gemma-2B (vocabulary size of 256,128 tokens/bytes) can yield 1GB of output logits per batch that must be copied across the PCIe bus to CPU RAM.

Our unified execution strategy consolidates this boundary; the relational filter predicate is evaluated on the TPU, as well as the LLM forward pass (semantic classification) and the final *SUM* aggregation/reduction. Consequently, no logits or intermediate arrays are copied back to the host. Only the final single scalar sum or the final generated tokens are copied back to CPU RAM, reducing D2H transfer data sizes from gigabytes to kilobytes. This consolidation completely removes PCIe bug bottlenecks and enables the massive throughput speedups demonstrated in Section ??.

4 PRELIMINARY RESULTS

We evaluated our AI query compilation prototype on a Google TPUEVM with four TPU v5e [9]) devices. For the initial benchmark, we processed a dataset of 49,664 rows to measure performance at scale and observe horizontal multi-device scaling behaviors.

Baseline and Evaluation Scope: For our baseline, we compare our unified compiled program against an eager mixture execution model (CPU for standard SQL and TPU for LLM inference). This baseline represents the standard architecture of contemporary AI Query Engines [5, 20, 21, 23], where standard relational query steps and data processing runs on the CPU host DBMS and the semantic operators/UDFs are run on hardware accelerators outside DBMS or even offloaded to an independent ML serving framework.

It is important to note that optimizing traditional SQL processing performance on the TPU hardware is not within the scope of this paper. Our compiler does not write customized, low-level TPU hardware kernels nor does it seek to build a highly optimized database engine on TPUs. Traditional SQL processing optimization on TPUs (or GPUs) is a separate, interesting area of research in itself, and our design is orthogonal to those efforts [4, 11, 12, 24–26, 29].

Having said that, our evaluation scope is strictly focused on demonstrating the feasibility, scalability and benefits of extra data-copy elimination via the unified execution path for hybrid AI queries. Our primary contribution is showing that we can compile a single program for such hybrid AI queries, which effectively

	Description	AI Query
Q1	Aggregation with semantic filtering	SELECT SUM(values_a) FROM movie_table WHERE AI_IF(reviewText, 'Is this review positive?') = 1
Q2	Aggregation with relational and semantic filtering	SELECT SUM(values_a) FROM movie_table WHERE values_a > 70.0 AND AI_IF(reviewText, 'Is this review positive?') = 1
Q3	Semantic transformation with relational predicate	SELECT AI_TRANSFORM(reviewText, 'One sentence summary of review:') FROM movie_table WHERE values_a < 40.0
Q4	Classification with cross join	SELECT A.reviewText, B.reviewText, AI_IF([A.reviewText, B.reviewText], 'Almost identical reviews?') FROM movie_table A CROSS JOIN movie_table B WHERE values_a < 40.0 AND A.id = B.id AND A.review_id < B.review_id
Q5	Aggregation across long output sequences with extra debug outputs	SELECT COUNT(*) FROM movie_table WHERE AI_TRANSFORM(reviewText, 'Rewrite this review as a detailed 3-page critical essay', debug=True) LIKE "%masterpiece%"

Table 2: Distinct AI queries. Q1-2 (aggregation with semantic filtering) are taken from *SemBench* for Rotten Tomatoes Critic Reviews and Movies datasets; Q3-4 are extended/synthesized AI queries for more rigorous testing; Q5 is an aggregation query with long output sequences and also with various debug information (e.g., full sequence logits to explain the LLM outputs)

eliminate the cross-boundary data movements and yield more performance optimization opportunities. Furthermore, by compiling the entire hybrid query as a single program, we make multi-device horizontal scaling fully automatic and easy via XLA’s GSPMD/*pmap*, sharding both relational arrays and model weights. Consequently, we do not compare our prototype system directly with highly customized GPU-based database systems.

Evaluation Queries and Datasets: To evaluate our approach on a more realistic AI query workload, we used a select analytics queries from *SemBench* [16]. In particular, we evaluate the three AI queries using the Rotten Tomatoes Critic Reviews and Movies datasets – the reviews table is joined with the movie details table on movie *id*, and we use *textit*audienceScore as a traditional filter column *values_a*. We originally focused on the Movies scenario with text-only semantic filtering (*AI_IF*). To evaluate our approach across a broader suite of operators, we synthesized a semantic mapping query (*AI_TRANSFORM*) on top of this text workload. While the benchmark includes out-of-the-box semantic mapping queries in other scenarios, those native workloads are heavily multi-modal (e.g., joining textual attributes with images). This would pose an orthogonal compiler complexities (e.g., using VLM [3]), which is beyond the scope of this paper and our preliminary evaluation. We compiled and evaluated five distinct AI queries from Table 2, at a scale of up to 49664 reviews.

4.1 Movie Review Analysis Performances

We evaluate aggregation and transformation AI queries (Queries 1-3 in Table 2) based on *SemBench*, that represent more realistic AI query workloads.

Our unified compiled execution approach demonstrates strong speedups on the semantic analytics workloads. For Query 1, we achieve a strong 5.34x latency speedup and 5.30x throughput speedup over the JIT-compiled baseline on dense records (low selectivity). The throughput gains remain highly robust even for hybrid

Table 3: SemBench Movie workload performance comparison on 4 TPU devices.

Query Type	Baseline Latency (ms)	Unified Latency (ms)	Throughput (rows/s)	Latency Speedup	Throughput Speedup
Q1: Semantic Filter	48124.34 ± 150.55	9017.40 ± 0.09	5508 ± 0	5.34x	5.30x
Q2: Hybrid Filter (> 70.0)	20590.13 ± 130.27	9017.13 ± 0.08	5508 ± 0	2.28x	5.31x
Q3: Semantic Map (< 40.0)	5344.06 ± 112.61	11388.57 ± 0.06	4361 ± 0	0.47x	4.29x

queries where a large fraction of records match the filter (2.28x latency speedup for Query 2). This gain is driven by the compiler’s ability to perform automatic multi-device hardware sharding (PMAP/GSPMD) to parallelize the LLM forward passes, which fuse and parallelize both SQL operator and AI semantic operators across available devices without manual rewriting the database and/or the ML serving engines. The baseline uses a single-process database engine on CPU, feeding data to separate TPU inference engine on TPUs; this is less efficient without extra parallelization and tuning efforts across the execution engines. In Section 4.4 we discuss how the unified program execution can be parallelized using the compiler’s auto-sharding feature, achieving linear horizontal scaling.

It is also important to note that the baseline CPU-TPU split execution runs faster for Q3. This is because the filter selectivity is high and leaves only 10.9% of the rows for LLM inference. This early data skipping is more advantageous running a static unified computation – where the filtering happens as part of the unified computation on TPUs with full data.

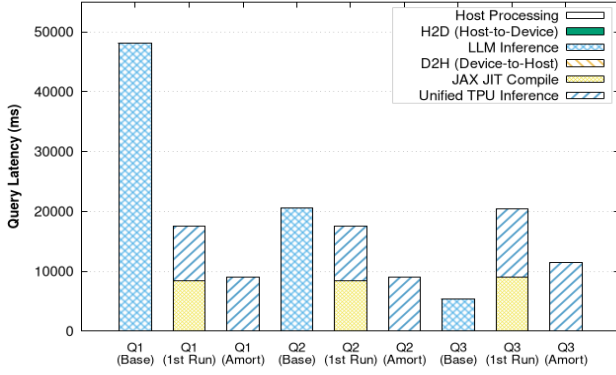


Figure 4: Latency breakdown for Queries 1-3 for the baseline (split execution) and the unified approach. Note that we compile once (1st Run) and re-use the compiled program to amortize the compilation cost across the subsequent micro-batches (Amort).

4.2 Impact of Filter Selectivity

The filter selectivity and how much data we are pushing to the downstream semantic operators for expensive LLM inferences governs the performance trade-offs between the baseline and the unified approach.

Table 4: Q3 performance across movie rating selectivity sweeps, $values_a < Thresh$.

Thresh	Baseline Latency (ms)	Unified Latency (ms)	Latency Speedup	Throughput Speedup
10.0	171.98 $\pm$ 92.64	11383.53 $\pm$ 0.14	0.02x	9.87x
30.0	2140.67 $\pm$ 97.35	11383.35 $\pm$ 0.06	0.19x	4.35x
50.0	11362.98 $\pm$ 108.22	11383.32 $\pm$ 0.06	1.00x	4.24x
70.0	27350.21 $\pm$ 131.81	11383.32 $\pm$ 0.06	2.40x	4.21x
90.0	43947.62 $\pm$ 153.13	11383.41 $\pm$ 0.14	3.86x	4.20x

We swept the filter threshold values on the semantic mapping query, Q3 (AI_TRANSFORM), illustrated in Table 4. We make an interesting observation where the high filter selectivity (Thresh=10.0) favors the baseline, split execution model. That is, we can eagerly skip most of the data for LLM inference computation, whereas the whole AI query JIT compilation needs to process the entire rows with a static micro-batch size. Notice the constant latency across different selection thresholds, driven by the static program shape and the fixed number of micro-batches over 49,664 rows.

Despite the latency penalty at high selectivity, the unified compiled execution achieves throughput speed-ups (4.2x to 9.8x rows/sec) as selection fraction increases (more rows to queue for LLM inference).

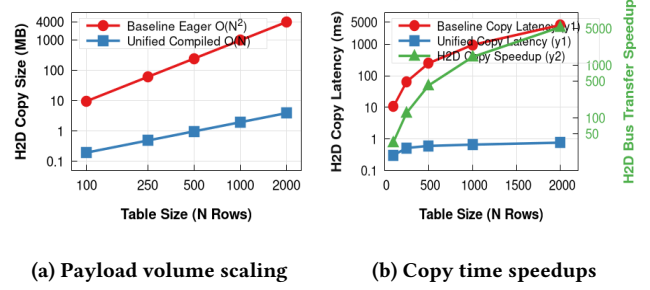


Figure 5: The cartesian (cross-join) Host-to-Device (H2D) data transfer overheads and speedups for Q4 execution

4.3 Latency Breakdown and Data Transfer Bottlenecks

Figure 4 illustrates the latency breakdown for Queries 1-3. Looking closely at the latency breakdown, we outline following key systems insights.

Sequential accelerator execution dominates the baseline: The overwhelming execution cost in the baseline is sequential LLM inference serving queue time on the TPU. The input data is streamed to TPUs as micro-batches (bsz=128) and the baseline incurs a lot of context switching overhead between CPU SQL database engine and TPU LLM inference (LLM Inference $\gg$ Unified TPU Inference). The unified JAX program, on the other hand, either loads the entire raw tabular data once or process the micro-batches without the expensive CPU-TPU synchronization. The compiler can also fuse SQL operators with LLM layers (e.g., scalar operations with embedding lookups or initial tokenization steps).

Data transfers are not primary latency bottlenecks for simple aggregation and transformation queries. The total data transfer overhead is negligible (0.2% of the baseline execution) for the single relation data that fits locally and with compact output formats (e.g., Yes/No classification results). While negligible, this PCIe transfer bottleneck can escalate dramatically with queries that require a large data transfer (H2D) and/or intermediate token embeddings or logits back to the host (D2H) for confidence-based soft filtering and even debugging. For instance, the cartesian (cross-join) query, Q4 from Table 2 for identifying almost identical movie reviews would requires sending $O(N^2)$ (N: relation size) input rows/tuples to the ML serving engine for the baseline, whereas only $O(N)$ for the unified approach. As illustrated in Figure 5, the cartesian H2D data transfer quickly becomes a system bottleneck; even for a small table size (N=2000), the transfer copy latency escalates to 4.06 seconds. By contrast, the dynamic data movement footprint remains strictly linear with the unified compiled approach. Rather than materializing duplicating review pairs on the host, the unified program requires a negligible 3.9 MB copy transfer size vs. 3.9 GB at N=2000.

LLM inference results can also potentially escalate the data transfer (D2H) bottleneck. The simple test queries Q1-3, returned argmax output tokens (e.g., Yes, No binary classification results) only; however, for analytics cases where raw distributions or logits are required for debugging/explaining the semantic operator decisions, the output payload size, thus the D2H data transfer latency can grow

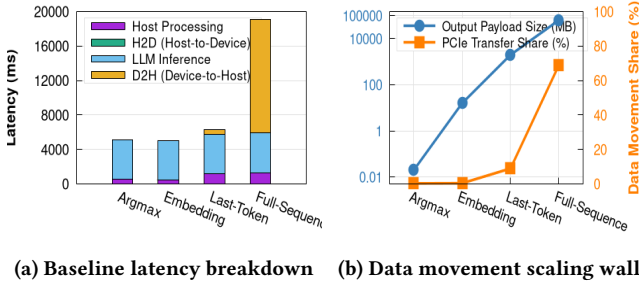


Figure 6: Device-to-Host (D2H) data transfer overheads and scaling by output format/structure for Q5 execution

significantly, as shown in Figure 6. Query 5 from Table 2 aggregates over long sequence outputs that contain a word `%masterpiece%`. As with the sequence length, the output formats (e.g., argmax tokens, latent vectors/embeddings, logits for the last token, full sequence logits) change the size of the payload drastically. The key challenge here is the host-side *LIKE* string match, that require moving all the transformation results back to the host for filtering. For the unified execution, we fuse and execute the string matching directly on local HBM, transferring the debug outputs for the matching essays only. Even without the extra debug outputs, the unified execution only needs to transfer the aggregation result, whereas the baseline still needs everything for the host processing of the aggregation.

4.4 Horizontal Scaling

This unified approach provides more than just the data movement latency elimination, unlocking new opportunities for global cross-boundary optimizations for AI queries. A major global optimization enabled by this unified program is unified, compiler-driven multi-device sharding. In traditional split AI Query Engine architectures, developers must manually write, calibrate, and synchronize two completely independent parallelization strategies: one for the CPU database engine (e.g., dynamic core-level partitioning) and another for the TPU/GPU machine learning engine (e.g., tensor-parallel or pipeline-parallel model sharding). This requires complex dynamic load balancing across the CPU-TPU boundary. By contrast, because our compiler translates both relational database scans and LLM inferences into a single unified tensor program, the compiler can globally shard both the relational data arrays and the neural computations concurrently and automatically via XLA’s GSPMD/PMAP. This is very convenient and also very effective, as we illustrate in Table 5. We evaluated the horizontal scaling capability of our system by sweeping the number of TPU devices from 1 to 4.

Table 5: Horizontal scaling performance across multiple TPU devices using Q3 with values_a < 50.0

Devices	TPU Time (ms)	Throughput (rows/s)	Speedup
1	45753.39 ± 0.04	1091 ± 0	1.00x
2	22878.20 ± 0.03	2182 ± 0	2.00x
4	11383.98 ± 0.07	4363 ± 0	4.00x

We replicate the same unified compiled program on all devices, but with different micro-batches across them. This enables near-perfect linear horizontal scaling, per the simple Aggregation example query. This demonstrates that the unified compiled model is extremely scalable, just like any LLM workloads with data parallelism, and fully prepared for massive distributed execution workloads. It is important to note that our preliminary evaluation looked at intra-node multi-device scaling. This overlooks the large-scale sharding as well as inter-node network overheads. We plan to investigate the multi-node scaling in the future.

4.5 Multi-Tenancy & Concurrent Query Bottlenecks

In real-world enterprise database systems, multiple users (tenants) execute queries concurrently. While specialized LLM serving engines (e.g., vLLM) employ highly engineered continuous batching and tensor parallelism to handle concurrent requests from multiple clients, our system compiles each query as a standalone JAX program/executable.

Table 6: Multi-tenant concurrent execution scalability.

Concurrent Tenants (K)	Average Latency (ms)	Aggregate Throughput (rows/s)	Throughput Scaling
1	3624.33 ± 0.03	1376 ± 0	1.00x
2	5435.86 ± 0.01	1377 ± 0	1.00x
4	9058.71 ± 0.14	1377 ± 0	1.00x
6	12681.62 ± 0.17	1377 ± 0	1.00x
8	16303.90 ± 0.41	1377 ± 0	1.00x

To evaluate the scalability of compiled AI queries under multi-tenant workloads, we spawned k concurrent client threads, each submitting an instance of the compiled *AI_IF* semantic filter query, where each query has to process 5000 rows. All threads shared the same on-device *Gemma-2B* parameters and invoked the same JIT-compiled program concurrently. We swept k from 1 to 8 and measured aggregate throughput (total rows processed across all clients divided by total elapsed time) and individual client latencies. Table 6 summarizes the results.

The experimental results reveal a bottleneck in concurrent execution: the aggregated throughput remains constant at exactly 1377 rows/sec, while average query latency scales linearly with the number of tenants. This is because the TPU execution queue is serialized and the JAX programs from different and concurrent queries must run one after the other (FIFO queue). To support the unified compilation paradigm in high-throughput enterprise DBMS, we need to support a runtime scheduler that can dynamically schedule and re-schedule different programs in the TPU queue to meet the SLO and increase the throughput, like continuous batching in vLLM.

5 CONCLUSION & RESEARCH ROADMAP

In this vision paper, we demonstrated the feasibility and potential performance gains of compiling hybrid AI queries as a whole into a single, unified JAX tensor program for execution on specialized accelerators (TPUs). By consolidating accelerator boundaries and

compiling a single unified program, we successfully amortize JAX JIT compilation overheads and enable compiler-driven multi-device sharding and optimizations for the CPU database and TPU ML inference engines (up to 4.00x scaling on 4 devices) on both synthetic benchmarks and *SemBench* workloads. To realize the full potential of this unified execution strategy, several key research challenges must be solved by the systems and database community:

Semantic-Aware Cost-Based Query Optimization: Traditional database optimizers rely on relational cost models (e.g., cardinality, selectivity) to choose execution plans. For the modern AI Query Engine, the optimizer must balance relational selectivity with LLM-based AI semantic operator costs. This cost-based AI Query optimization pose a similar challenge as the inlining vs. outlining trade-offs for traditional UDF optimizer [1]. Instead of deciding whether to inline procedural CPU-based UDFs, an AI Query optimizer may choose between the unified compiled execution and the split execution strategies.

Dynamic Shape Compilation: JAX/XLA compilers rely heavily on static tensor shapes to generate highly optimized TPU/GPU machine executables, and to avoid frequent re-compilation. However, real-world AI queries can involve input data of highly varying lengths/sizes, requiring dynamic input shapes. The current prototype system pads inputs to static maximum sequence lengths, which waste TPU memory and compute. Research is needed to enable efficient dynamic-shape tracing for the hybrid AI query workloads.

Multi-Query Concurrent Execution: In high-throughput enterprise databases, multiple queries run concurrently. The current stack highly optimizes for static graph and shape specialization, taking one executable at a time. This limits the performance with multi-tenancy, where more dynamic scheduling and execution, possibly with token interleaving across independent concurrently queries are desired [28].

Multi-Modal Compilation: Future AI queries will involve multi-modal operators processing text, image, audio, video, etc. More work is needed to support a wide range of operators for efficient compilation, but also the increasing sizes and complex model architectures raise sharding and partitioning challenges.

Dynamic TPU Data Skipping: One of the challenges in the unified program JIT compilation is the lack of dynamic, on-device data skipping. As discussed in Section 4.2, the baseline, split execution outperforms the unified execution with high selectivity filter predicates. The static tracing and compilation forces full table scan regardless selectivity. Research is needed to solve this compile vs. skip trade-off.

ACKNOWLEDGMENTS

We would like to thank Hank Levy, David Culler, Greg Ganger, Arvind Krishnamurthy and Brian Suchy for their valuable feedback and discussion.

REFERENCES

- [1] Samuel Arch, Yuchen Liu, Todd C Mowry, Jignesh M Pate, and Andrew Pavlo. 2026. Partial UDF Inlining. *ACM SIGMOD Record* 55, 1 (2026), 74–83.
- [2] Matthias Boehm, Matteo Interlandi, and Chris Jermaine. 2023. Optimizing tensor computations: From applications to compilation and runtime techniques. In *Companion of the 2023 International Conference on Management of Data*. 53–59.
- [3] Florian Bordes, Richard Yuanzhe Pang, Anurag Ajay, Alexander C Li, Adrien Bardes, Suzanne Petryk, Oscar Mañas, Zhiqiu Lin, Anas Mahmoud, Bargav

- Jayaraman, et al. 2024. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247* (2024).
- [4] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*.
- [5] Yeounoh Chung, Rushabh Desai, Jian He, Yu Xiao, Thibaud Hottelier, Yves-Laurent Kom Samo, Pushkar Khadilkar, Xianshun Chen, Sam Idicula, Fatma Özcan, et al. 2026. 100x Cost & Latency Reduction: Performance Analysis of AI Query Approximation using Lightweight Proxy Models. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data (SIGMOD '26)*. ACM.
- [6] Yeounoh Chung, Thibaud Hottelier, Cosmin Arad, Brenton Milne, Per Jacobsson, Sam Idicula, Fatma Özcan, and Alon Halevy. 2026. Architecting the AI-Powered Agentic Data Cloud. *IEEE Data Engineering Bulletin* 49, 1 (March 2026), 22–31.
- [7] Roy Frostig, Matthew James Johnson, and Chris Leary. 2019. Compiling machine learning programs via high-level tracing. In *SysML conference 2018*.
- [8] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (2024), 20–32.
- [9] Google Cloud. 2026. TPU v5e. Google. <https://docs.cloud.google.com/tpu/docs/v5e> Accessed: 2026-05-11.
- [10] Google Cloud. 2026. Vertex AI: Google Cloud’s Unified Machine Learning Platform. <https://docs.cloud.google.com/vertex-ai/docs>. Accessed: May 8, 2026.
- [11] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877* (2022).
- [12] Pedro Holanda and Hannes Mühleisen. 2019. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.
- [13] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*. 1360–1374.
- [14] Wentao Huang, Mian Lu, and Kian-Wei Tan. 2026. Hash Joins Meet CXL: A Fresh Look. In *Proceedings of the 2026 Conference on Innovative Data Systems Research (CIDR '26)*. <https://www.vldb.org/cidrdb/papers/2026/p1-huang.pdf>
- [15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
- [16] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2026. SemBench: A Benchmark for Semantic Query Processing Engines. *arXiv:2511.01716 [cs.DB]*. <https://arxiv.org/abs/2511.01716>
- [17] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.
- [18] Guanghua Li, Hao Zhang, Xibo Sun, Qiong Luo, and Yuanyuan Zhu. 2024. Ten-graph: A tensor-based graph query engine. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4571–4584.
- [19] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M Maas, Momin Al-Ghosen, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, et al. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4518–4531.
- [20] Paweł Liskowski, Benjamin Han, Paritosh Aggarwal, Bowei Chen, Boxin Jiang, Nitish Jindal, Zihan Li, Aaron Lin, Kyle Schmaus, Jay Tayade, et al. 2026. Cortex AISQL: A Production SQL Engine for Unstructured Data. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data (Companion) (SIGMOD '26)*. ACM, 421–434. <https://doi.org/10.1145/3788853.3803093>
- [21] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Bailie Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A declarative system for optimizing ai workloads. *arXiv preprint arXiv:2405.14696* (2024).
- [22] Vasilis Mageirakos, Joel André, Marko Kabić, Bowen Wu, Yannis Chronis, and Gustavo Alonso. 2026. To GPU or Not to GPU: Vector Search in Relational Engines. *arXiv preprint arXiv:2605.15957* (2026). [arXiv:2605.15957 \[cs.DB\]](https://arxiv.org/abs/2605.15957) <https://arxiv.org/abs/2605.15957>
- [23] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment (PVLDB)* 18, 11 (2025), 4171–4184. <https://doi.org/10.14778/3749646.3749685>
- [24] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

- [25] Peter Benjamin Volk, Dirk Habich, and Wolfgang Lehner. 2010. GPU-Based Speculative Query Processing for Database Operations.. In *ADMS@ VLDB*. 51–60.
- [26] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.* 7, 11 (2014), 1011–1022.
- [27] Wm A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.
- [28] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*. 521–538.
- [29] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. 2013. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1374–1377.