Benchmarking Native In-Database TPCx-AI at 100 Terabytes in Ocient Hyperscale Data Warehouse: An Eight-Use-Case Study of Classical and Statistical ML on Relational Primitives

Jason Arnold
Ocient
Chicago, USA
jarnold@ocient.com

Neesh Dahiya
Ocient
Chicago, USA
ndahiya@ocient.com

Knut Stolze
Ocient
Jena, Germany
stolze@ocient.com

ABSTRACT

The growth of enterprise data has created a scalability gap in machine learning (ML) workflows. As data volumes climb into the hundreds of terabytes, extracting raw data to external processing frameworks such as Apache Spark imposes increasing costs in serialization latency, network transfer, and data governance. Audited TPCx-AI results, the standard benchmark for end-to-end ML pipelines, have been limited to scale factor 3,000 (3 TB) to date.

This paper presents the architecture and performance of the Ocient Hyperscale Data Warehouse executing eight of the ten TPCx-AI use cases at scale factor 100,000, an aggregate generated dataset size of 100 Terabytes. To our knowledge, this is the largest reported TPCx-AI execution, more than an order of magnitude beyond the largest audited result. Two use cases that require deep-learning model families our engine does not yet implement natively are omitted. For the eight executed use cases, we demonstrate linear or sub-linear scalability across six scale factors (1 GB to 100 TB), meeting the TPCx-AI accuracy target at every scale on every use case except one minority-class classification case whose threshold is also missed by our algorithm-matched Spark implementation, indicating that the gap is intrinsic to the synthetic dataset rather than to either engine. Our goal is to provide an existence proof that a single relational engine can execute classical and statistical ML pipelines at scale on high-end, off-the-shelf enterprise hardware.

We detail a database architecture designed to feed query operators directly from NVMe drives to CPU registers via userspace polling. We describe a memory management system using 128 KB fragments within 1 GB hugepages, and how we achieve purely relational execution of iterative algorithms utilizing native matrix data types and operators in standard SQL: no UDFs, no embedded R/Python runtime, and no separate linear-algebra DSL are needed. Furthermore, we discuss a native C++ matrix library and an LLVM-based JIT compiler that translates decision tree models directly into machine code. To position the architectural choices, we report an algorithm-matched comparison against Apache Spark across four scale factors from 100 GB to 100 TB.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

TPCx-AI, In-Database Machine Learning, Hyperscale, SPDK, LLVM JIT, Relational Databases

PVLDB Reference Format:
Jason Arnold, Neesh Dahiya, and Knut Stolze. Benchmarking Native In-Database TPCx-AI at 100 Terabytes in Ocient Hyperscale Data Warehouse: An Eight-Use-Case Study of Classical and Statistical ML on Relational Primitives. PVLDB, 19(12): 4143 - 4155, 2026.
doi:10.14778/3827998.3828022

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Ocient/vldb2026-tpcxai.

1 INTRODUCTION

The proliferation of machine learning (ML) has fundamentally altered the requirements for modern data management systems. Historically, data warehouses were optimized for aggregation and reporting, the workloads were characterized by high selectivity and predictable access patterns. However, the rise of predictive analytics requires systems to support fundamentally different workloads: they involve massive, sequential scans of raw data, complex feature engineering pipelines involving text processing and joins, and iterative training algorithms that must pass over the entire dataset multiple times.

As enterprise data volumes grow, the conventional "extract, transfer, train" paradigm has well-known scaling costs. Moving 100 TB of data from a data warehouse to a separate ML cluster (e.g., Apache Spark [42] or a specialized GPU farm) takes hours of wall clock time even on 100 Gbps networks. On top of that, shipping training data out of a governed environment introduces compliance and audit concerns. We do not claim external ML frameworks are obsolete (they remain the right tool for many workloads, particularly deep learning), but for classical and statistical ML on warehouse-resident data, an *in-situ* alternative [5] becomes increasingly attractive.

While "In-Database ML" is an established concept [22], existing implementations often struggle with the "Memory Wall" [40] when scaling beyond terabytes. Standard database buffer pools are ill-suited for the massive, sequential scans required by ML training. When the working set exceeds the buffer pool size, the system thrashes, spending more CPU cycles managing cache eviction than actually processing data.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828022

The architectural challenges this paper addresses are therefore concrete. First, the system must sustain NVMe-saturating throughput on full-table scans for hundreds of terabytes without page-cache thrashing. Second, it must manage tens of gigabytes of intermediate state per node without TLB and fragmentation penalties. Third, it must support iterative training in a shared-nothing distributed engine via single-aggregation-per-iteration synchronization that reuses the engine's existing aggregate operator (the IRLS pattern of Section 5.1). Fourth, it must compile tree-based inference fast enough that scoring billions of rows is not the long pole.

In this paper, we present the architecture of Ocient's native ML engine, designed specifically to overcome these limitations. We validate this architecture by executing the TPCx-AI benchmark [38] at scale factor (SF) 100,000. We loaded the full 100 TB dataset, including image and audio data stored as VARBINARY columns. We executed eight of the ten use cases (UC1, UC3, UC4, UC5, UC6, UC7, UC8, UC10) and omit UC2 and UC9, which use deep learning for audio-to-text and facial recognition respectively; these are model families our engine does not yet support natively (Section 2.2). This benchmarking effort aligns with industry efforts like MLPerf [30] to provide objective standards for ML system performance. The largest audited TPCx-AI result to date is at SF=3,000 (3 TB) [37]; vendor announcements have reported 1 TB-scale runs on specific hardware [9]. Our SF=100,000 execution extends this by two orders of magnitude on the eight use cases we cover.

All workload code (Java drivers, per-UC SQL templates, algorithm-matched PySpark drivers, and the extracted result CSVs that populate the tables in this paper) is released as a companion artifacts repository at https://github.com/Ocient/vldb2026-tpcxai.

Our primary contribution is a detailed architectural analysis of how to build a database engine capable of native ML at this scale. We organize the paper around six architectural pillars (Section 3–Section 6): bufferless SPDK-based I/O [36]; the hugepage fragment-pool allocator; lease-based push-based shuffle backpressure; the native C++ matrix engine without BLAS/LAPACK [2, 20]; pure relational iterative training over first-class matrix column types; and the LLVM-IR [19] JIT path for tree-based inference.

The remainder of the paper is organized as follows. Section 2 introduces the TPCx-AI benchmark, the two omitted use cases (UC2 and UC9), and the Ocient data warehouse. Section 3 through Section 6 cover the six architectural pillars listed above. Section 7.1–Section 7.7 describe the three use cases that receive a full architectural treatment (UC1 relational K-Means; UC3 grouped-CTE matrix INVERSE for per-key regression; UC8 pivoted feature engineering with sparsity-aware binning), with the remaining executed use cases summarized in the companion artifacts repository. The experimental section reports the SF=1,GB through SF=100,TB sweep on Ocient, the model-quality results (Section 9.3), and the algorithm-matched Spark comparison (Section 9.4). Section 10 positions the work relative to prior in-database-ML systems, Section 11 lists active directions, and Section 12 concludes.

2 BACKGROUND

2.1 The TPCx-AI Benchmark

The TPCx-AI benchmark is the industry standard for measuring the performance of end-to-end AI/ML pipelines [38]. Unlike micro-benchmarks that test only the training phase of a specific algorithm, TPCx-AI evaluates the entire data lifecycle. It defines ten use cases (UC) ranging from customer segmentation to fraud detection.

For this study, we targeted scale factor (SF) 100,000. This corresponds to a total generated dataset size of 100 Terabytes across all defined use cases. At this scale, the benchmark ceases to be a test of algorithm convergence and becomes a rigorous stress test of the underlying system's I/O throughput, memory management, and distributed coordination capabilities. We successfully executed eight of the ten use cases: UC1, UC3, UC4, UC5, UC6, UC7, UC8, and UC10. The two omitted use cases are discussed below.

2.2 Omitted Use Cases (UC2, UC9) and Future Path

Two of the ten TPCx-AI use cases require deep-learning model families that Ocient does not yet implement natively: UC2 (audio-to-text on customer-service recordings, typically realized with a recurrent [16] or transformer-based [39] acoustic model) and UC9 (facial recognition on customer images, typically realized with a convolutional neural network [21]). Ocient's native ML kernel currently provides feed-forward (multi-layer perceptron) networks. We attempted to express UC2 and UC9 with the relational primitives and statistical models available today and were unable to meet the benchmark's accuracy thresholds with those building blocks; we therefore chose to report no result for these two cases rather than report a result that fails the specification.

We see two realistic future paths to closing this gap. The first is to extend the native ML kernel with convolutional and sequence-model layer types so that UC2 and UC9 are expressible in the same purely-relational style as the other eight use cases. The second is the Java and Python user-defined-function and stored-procedure infrastructure currently under development in the engine; with UDF/stored-procedure support, a pre-trained external model (for example, an off-the-shelf acoustic model [15] or face-embedding model [32]) can be invoked as a SQL-callable function, with the database supplying the rows-to-tensor batching and result aggregation. The second path is the more practical near-term route because the same infrastructure covers both: stored procedures host the training loop and UDFs serve the trained model. The first path is the longer-term answer for performance reasons; native layer types let the planner and matrix engine see the model end-to-end, the same way they see the other eight use cases today.

2.3 The Ocient Data Warehouse

Ocient [34] is a distributed, shared-nothing relational database system engineered for extreme-scale analytical workloads. This architecture builds upon the foundational principles of parallel database systems like Gamma [10] and the concept of "Shared Nothing" [35], updated for the NVMe era. It separates the **SQL Node** (Control Plane) from **Foundation Nodes** (Storage and Compute Plane) (Figure 1).

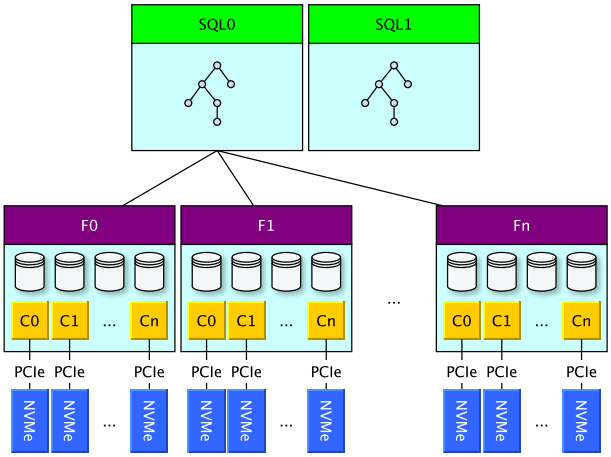

Figure 1: Ocient Architecture: SQL Nodes (e.g. SQL0 and SQL1) manage control flow while Foundation Nodes (F0, ..., Fn) execute heavy compute adjacent to NVMe storage.

SQL Node: This node handles query parsing, optimization, and orchestration. It maintains the global system catalog and serves as the entry point for client connections. Crucially, for ML workloads, this node hosts the "ML Coordinator," which is responsible for managing the state of iterative training algorithms.

Foundation Nodes: These nodes are the workhorses of the cluster. They store data on NVMe drives and execute distributed query operators.

3 SYSTEM ARCHITECTURE: BREAKING THE MEMORY WALL

Scaling machine learning to 100 TB requires a fundamental rethinking of how data moves from persistent storage to the CPU. The "Memory Wall" [40] refers to the growing disparity between processor speed and memory bandwidth/capacity. In the context of 100 TB datasets, the primary bottleneck is not floating-point operations per second (FLOPS), but rather the rate at which data can be fed into the ALUs.

3.1 Bufferless, Kernel-Bypass I/O

A defining characteristic of the Ocient architecture is the absence of a traditional database buffer pool for scan-heavy workloads. In standard database systems (e.g., PostgreSQL, Oracle [26]), the buffer pool serves to cache hot pages in RAM. However, for ML training workloads at 100 TB, the access pattern is almost exclusively massive, sequential full table scans. A buffer pool in this context creates cache thrashing and CPU overhead for LRU management. [1]

We leverage the Storage Performance Development Kit (SPDK) [36] to perform kernel-bypass I/O.

[1]Storage layer details, including the compressed columnar segment format, are orthogonal to the architectural points in this section and are not central to the ML workloads we describe.

Userspace Polling: Modern asynchronous interfaces like `io_uring` [4] amortize per-I/O syscall cost through shared submission/completion ring buffers, but the kernel still owns the NVMe driver, interrupt-driven completion, and the block-device stack. SPDK takes the kernel out of the I/O path entirely by dedicating specific CPU threads to drive NVMe submission/completion queues directly via userspace MMIO. Idle polling threads sleep and periodically check for new work, but switch to hot polling when requests are active to minimize latency to hardware limits.

Zero-Copy Path: Data is transferred via Direct Memory Access (DMA) from the SSDs into the destination memory buffers where it needs to be used next. This ensures that data is moved across the memory bus only once without involving copies via kernel space.

3.2 Memory Management: The Hugepage Fragment Pool

When performing in-database processing at this scale, the system requires efficient memory management. Standard OS allocators (`malloc()`) often cause fragmentation and suffer from TLB (Translation Lookaside Buffer) misses when dealing with multi-gigabyte intermediate structures (sort runs, hash tables, matrix accumulators) on the 2.3 TB-RAM Foundation servers used for our evaluation (Section 9.1), where a 4 KB-page TLB cannot cover even a single working set.

To solve this, we implemented a custom allocator: the **Hugepage Fragment Pool**. At system boot, we pre-allocate roughly two-thirds of system RAM as 1 GB Hugepages. Within the database kernel, these hugepages are sliced into 128 KB fragments. These fragments are merely contiguous 128 KB pieces of memory; we build custom data structures (vectors, deques, hash maps, hash sets, bloom filters) directly on top of these fragments. We also have a reader/writer API that makes multiple fragments look like a larger contiguous chunk of memory. This allows us to operate on logically contiguous structures without requiring the operating system to find contiguous physical pages. Note that currently, matrix objects do not yet route through this abstraction (the small-matrix fast path is contiguous up to 128 KB; larger matrices materialize through a fragmented path with a per-matrix cap of 512 MB). All intermediate matrices in the workloads reported here are small (3×3 normal-equations matrices for the per-group multiple linear regressions in UC3 and UC10, and at most ≈40×40 for UC6's polynomial-feature MLR) and stay on the contiguous fast path.

3.3 Push-Based Network Backpressure

A 100 TB workload puts significant pressure on the network fabric, particularly during shuffle-heavy operations such as the 68-column `GROUP BY` pivot in Use Case 8 (one grouping key with 68 `SUM(CASE)` aggregates) (Section 7.7) or the customer-level join across `lineitem`, `orders`, and `order_returns` in Use Case 1 (Section 7.1). To manage backpressure during these shuffle phases, we implemented a **Push-Based Lease** mechanism.

The receiving node grants "leases" to sending nodes. A lease represents a permission to send N 128 KB fragments of data.

(1) The sender transmits data and decrements its active lease.

(2) If the lease reaches zero, the sender must pause transmission to prevent overwhelming the receiver.
(3) To maintain high throughput and prevent stalling, the receiver asynchronously and proactively issues new leases as its local ingestion buffers clear. A newly received lease supersedes the existing lease, allowing data to flow continuously.

This mechanism ensures that a fast sender cannot overwhelm a slow receiver. While the hardware supports 100 Gbps, saturating this bandwidth via kernel-based TCP/IP is difficult. We observe peaks of 60–70% utilization during specific query phases, using multiple `libuv` threads per server [23].

3.4 Resilience and Operational Robustness

The query execution engine follows a fail-stop model [31]: we do not support speculative execution or mid-query fault tolerance. If an entire node is lost during a query, the query aborts and must be restarted. We adopted this trade-off deliberately, and the empirical data from our deployments supports it.

Failure Granularity is Mostly Sub-Node. Hardware failures at the cluster scale are statistically inevitable, but the failure modes that matter for query progress are far rarer than the raw component failure rate suggests. The storage layer uses erasure coding across the NVMe drives in the cluster. A query resolves its read targets at start time, so a drive that fails mid-query fails that query; subsequent queries rebuild from parity transparently on first read. Drive failures (the most common failure mode in our experience) therefore cost at most a single-query restart, not a cluster-level disruption. DDR5 RDIMM faults, the next most common mode, are typically detected as correctable errors that have no functional impact until the affected DIMM is replaced. NIC and motherboard failures, which *are* fatal to in-flight queries, have been observed roughly once per decade across hundreds of identical lab servers operating continuously over the past ten years.

Empirical Stability of Long Runs. The benchmark sweep reported in this paper (six scale factors from 1 GB to 100 TB) executes weekly as part of our continuous-quality testing on this exact 8-node Foundation cluster. The 100 TB run is dominated by UC8 training at approximately 34 minutes; no other individual training query exceeds 5 minutes, and the full 100 TB benchmark (preprocessing plus training plus serving) completes in approximately 78 minutes of wall-clock time (the cumulative sum of times reported in Table 1, totalling 4,701 s). Across these weekly runs we have not observed a node-fatal failure during a benchmark execution, which is consistent with the manufacturer-reported MTBF of the components: 2 million hours for the Solidigm D7-P5520 NVMe drives, and multi-million-hour figures for the Mellanox ConnectX-6 Dx NICs and Samsung DDR5 RDIMMs.

Why Fail-Stop is Acceptable Here. A mid-query fault-tolerance scheme (checkpointing intermediate state, lineage replay [42], or the speculative recovery used in long-running batch systems) is not free: it adds memory and I/O overhead to every query, complicates the optimizer, and constrains operator design. We chose to spend that complexity budget on raw throughput instead, and the expected restart cost is low in practice: a query that fails mid-run is restarted at the granularity of that single query, and the longest such query in the 100 TB sweep is well under the duration of the full sweep. Deployments that require stricter availability than fail-stop can provide configure multiple Ocient clusters in active-active across data centers at the application layer, so a cluster-level outage does not interrupt the workload; that cluster-redundancy approach is independent of the per-query-engine design choices discussed here.

4 NATIVE MATRIX ENGINE INTERNALS

A critical component of our ML engine is the implementation of linear algebra primitives directly within the database kernel. Rather than relying on external libraries like LAPACK [2] or BLAS [20], which would introduce linking complexities and memory management overhead, we utilize a native engine optimized for the Ocient environment. This allows strict control over memory usage, tight integration with the database's error handling, and avoidance of unnecessary data copying.

4.1 Implementation of Decompositions

Key decompositions were engineered from the ground up to ensure numerical stability.

4.1.1 LUP Decomposition with Partial Pivoting. For matrix inversion and solving linear systems, we utilize LUP decomposition. We use partial pivoting [12] to ensure numerical stability. The implementation tracks row exchanges to compute the determinant of the permutation matrix P. This decomposition is then used to solve $Ax = b$ for each column of the identity matrix using Crout's reduction [12].

4.1.2 Eigenvalue Solvers for PCA. For Principal Component Analysis, we require the eigenvalues of the covariance matrix. We execute the QR algorithm with Hessenberg reduction [12]. The process involves reducing the matrix to an upper Hessenberg form using Householder reflections, followed by iteratively applying Givens rotations to drive the sub-diagonal elements to zero using the Wilkinson shift to accelerate convergence.

4.2 Memory Ownership and Zero-Copy

Our matrix classes are designed to wrap existing memory buffers provided by upstream operators.

Non-Owning Mode: When operating on data streamed from disk, the matrix object wraps the existing byte pointer provided by the upstream operator. This is a true zero-copy construction.

Owning Mode: When storing intermediate results (like the X^TX accumulator), the matrix owns its backing memory and currently uses heap-backed storage. Migrating owning matrices to the hugepage fragment pool is a planned engine change; in the matrix shapes used by the workloads in this paper (3×3 normal-equations matrices for UC3 and UC10; at most ≈40×40 for UC6) the heap allocation is small enough that the distinction is not on the critical path.

5 NATIVE ML PRIMITIVES: PURE RELATIONAL ITERATION

Prior in-database ML systems have generally taken one of two paths to express iterative training. The library-of-UDFs path (e.g., MADlib [14]) implements each algorithm as a hand-written C aggregate function: fast, but opaque to the query optimizer and locked to a fixed catalog of algorithms. The DSL path (e.g., SystemML [6], exposing DML scripts that a separate compiler lowers to a Spark/MR plan) preserves optimizer visibility but requires the user to learn a new linear-algebra language and the engine to maintain a parallel compilation stack alongside its SQL planner. We take a third path: we lift matrix and vector to first-class column types in standard SQL. Matrix-valued expressions (`x_vec * TRANSPOSE(x_vec)`, `INVERSE(H)`) and matrix-valued aggregates (`SUM(H_part)`) flow through the same planner, optimizer, and execution engine as any other SQL expression. Iterative training is therefore not a special operator: it is a sequence of ordinary CTE queries, generated and submitted by a lightweight C++ coordinator that holds only the model parameters between iterations.

5.1 Case Study: Logistic Regression via Relational IRLS

To train Logistic Regression models, the system requires an iterative optimization process to minimize a loss function. Traditional *stochastic* gradient methods can be parallelized on shared-nothing clusters via parameter servers, AllReduce, or asynchronous updates, but each of these requires either a separate ML system layered on top of the database or a SQL-foreign synchronization primitive that the planner cannot reason about.

Therefore, our architecture is optimized for deterministic global updates computed over massive local data chunks. For Logistic Regression, we utilize Iterative Reweighted Least Squares (IRLS) based on the Newton-Raphson method [28]. IRLS calculates a deterministic global gradient and Hessian for the entire dataset per epoch. This allows foundation nodes to independently process massive local data chunks at NVMe speeds, requiring only a single, highly efficient distributed aggregation phase per iteration over the cluster's network fabric. Because IRLS updates parameters using the full dataset gradient and Hessian per step, it converges rapidly (often under 20 epochs) and translates directly to SQL aggregations. We note that IRLS itself is a textbook algorithm, used in MADlib among other systems; the contribution here is not the optimization method but the way it is expressed and executed: as a single relational query over first-class matrix types, not as a sequence of UDF calls or a compiled DML script.

The ML Coordinator manages the model coefficients (w). During each epoch, the Coordinator injects the current weights into a Common Table Expression (CTE) query that leverages Ocient's native matrix math types and aggregate functions to compute the Newton step update ($\Delta w = H^{-1} \cdot \nabla$).

Listing 1: A single iteration of IRLS computed entirely via standard relational operators and native matrix functions.

```
WITH VECTORIZED AS (
  -- x_vec represents the feature vector with bias
  SELECT {{x1}, {x2}, {1.0}} AS x_vec, yd,
         (w1*x1 + w2*x2 + w_bias*1.0) AS z
  FROM SCALED_DATA
),
PROBS AS (
  -- Calculate sigmoid probability
  SELECT x_vec, yd, 1.0 / (1.0 + EXP(-z)) AS p
  FROM VECTORIZED
),
CALC AS (
  -- Calculate Hessian (H_part) and Gradient (grad_part)
  SELECT ((p * (1.0 - p)) + 1.0e-12) * (x_vec * TRANSPOSE(x_vec))
         AS H_part,
         (p - yd) * x_vec AS grad_part
  FROM PROBS
),
AGG AS (
  -- Distributed parallel aggregation over matrices!
  SELECT SUM(H_part) AS H,
         SUM(grad_part) AS grad
  FROM CALC
),
SOLVED AS (
  -- Final matrix inversion to find delta weights
  SELECT INVERSE(H + (identity_matrix(3)*1.0e-8)) * grad AS delta_w
  FROM AGG
)
SELECT delta_w[1, 1], delta_w[2, 1], delta_w[3, 1]
FROM SOLVED;
```

5.1.1 Matrix Aggregation at Scale. Listing 1 illustrates the paradigm shift. The `CALC` block constructs an $N \times N$ Hessian matrix component and an $N \times 1$ gradient vector for *every single row* in the dataset. Because our execution engine supports true matrix types, the `AGG` block utilizes the standard SQL `SUM()` function to aggregate millions of matrices across the Foundation Nodes into a single global Hessian (H) and Gradient ($grad$) at the Control Plane.

Finally, the `SOLVED` block executes the `INVERSE()` function directly in the query. The C++ state machine retrieves this single-row result, updates the internal weights, and immediately issues the query for the next iteration until convergence. By lifting the linear algebra directly into the SQL standard library, we bypass the need for any "black box" ML operators in the execution plan.

5.2 Generalized Nonlinear Regression

The same coordinator pattern generalizes to arbitrary nonlinear models $y = f(x, \theta)$. The user supplies a custom function and loss; the coordinator parses both into an AST, derives analytical gradients via symbolic differentiation (or injects centered finite differences for non-differentiable subexpressions), and dispatches to the appropriate optimizer: Levenberg-Marquardt [25] for least-squares loss (Jacobian materialized once per epoch as a matrix-valued aggregate, damping injected into `INVERSE`), or full-batch Adam [17] for custom losses (one distributed `SUM(grad)` per epoch followed by an in-memory momentum/velocity update on the control plane). Per-epoch cost is dominated by a single distributed scan; sub-epoch sample queries (e.g. `LIMIT 1024`) support cheap step-size search.

6 JIT COMPILATION VIA LLVM

For inference tasks (UC8), evaluating decision trees via interpreted `CASE` statements is computationally inefficient. A single deep decision tree (UC8 uses `maxDepth=13`, yielding up to several thousand internal split nodes) evaluated row-by-row via an abstract syntax tree (AST) walker suffers from significant branching overhead and poor instruction cache locality; tree ensembles such as Random

Forests amplify this by a factor equal to the number of trees in the ensemble [41].

We implemented a Just-In-Time (JIT) compiler using the LLVM infrastructure to accelerate this process.

6.1 LLVM IR Generation

The compilation process translates the hierarchical model structure (represented via Protobuf [8]) directly into LLVM IR. We map each node in the decision tree to a set of LLVM Basic Blocks. Each JIT-compiled decision tree is generated as a distinct function within the execution engine.

6.1.1 Argument Handling and Pruning. Machine learning models often possess high dimensionality, resulting in a large number of potential input features. Passing these as a generic array introduces overhead. Instead, we utilize a `function` class that is templated on the number of arguments, allowing us to generate a function signature with explicit, typed arguments. Furthermore, a decision tree path may not utilize all available features, especially when the tree depth is shallower than the feature vector size. As an optimization, we analyze the tree structure at plan compilation time. We then generate a JIT-compiled function to accept only the features actually referenced by the logic, discarding unused inputs.

6.1.2 Constraints and Performance. The logic involves generating comparison instructions for each split in the tree. If the condition is true, control flow jumps to the "Then" block; otherwise, it jumps to the "Else" block. Currently, the JIT path supports standard comparison operators but does not yet support `IS NULL` checks. Models that require null-aware branching fall back to the interpreted scoring path; this preserves correctness at the interpreted-path performance level. We plan to extend the JIT path itself by passing a null-indicator bitmap to the generated function and hoisting the eligibility check into the SQL-level validator so the planner picks the path at parse/validate time.

We utilize the `llvm::ExecutionEngine` with aggressive optimization levels to optimize the generated machine code. The empirical signature of this is that the UC8 serve phase, which scores roughly 5.7 billion trip rows (one row per pivoted order) through a depth-13 decision tree at SF=100,000, completes in 38 s (Table 1); the interpreted-AST path was the long pole on UC8 serve before the JIT lowering was added.

7 IMPLEMENTATION DEEP DIVE: COMPLEX USE CASES

This section walks through the eight TPCx-AI use cases we executed. UC1, UC3, and UC8 receive a full Specification / Why-it-maps / Implementation treatment because each illustrates a distinct architectural pattern; UC4, UC5, UC6, UC7, and UC10 receive a single descriptive paragraph because they reuse the same primitives in less elaborate ways. Full per-UC SQL templates for every use case live in the companion artifacts repository [3]. The recurring pattern across all eight is that a small set of engine capabilities (aggregation, join, window functions, regex/array primitives, native matrix types, JIT inference) suffices to cover the breadth of classical and statistical ML in TPCx-AI.

7.1 UC1: Customer Segmentation via Aggregation

Specification. UC1 partitions customers into k behavioral segments using K-Means clustering. Inputs are the order, lineitem, and return tables; the model must train on per-customer summary features (order frequency, return ratio). There is no accuracy threshold (clustering is unsupervised) but the spec mandates k centroids be produced.

Why this maps cleanly. The dimensionality of the K-Means input is bounded by the number of distinct customers, which is small (~31 million at SF=100,000), regardless of how much raw transaction data exists. The expensive part of the pipeline is therefore a fixed-shape relational aggregation, and the ML training that follows is comparatively cheap. This is exactly the regime where parallel aggregation in a database has a significant structural advantage over an external ML system.

Implementation. The raw data at our target scale is approximately 36 billion `lineitem` records, 5.7 billion `orders` records, and 2 billion `order_returns` records. We first aggregate to the customer level (approximately 31 million unique customers), calculating per-customer features (frequency, return ratio) plus min-max-scaled versions of each, in a single CTE that joins the per-customer aggregate to a cluster-wide `(min, max)` aggregate. Because K-Means then runs on the aggregated customer table rather than the raw lineitems, training time stays remarkably stable across scales: 58 s at 1 GB, 65 s at 100 TB. Section 9.2 discusses the scaling implication.

7.2 UC3: Per-Key Regression Expressed as a Single CTE

Specification. UC3 forecasts weekly sales by Store and Department. The accuracy metric is mean squared logarithmic error (MSLE) against held-out future weeks; the target is 5.4. A natural decomposition is one AR(2) time-series model per (store, department) pair.

Why this maps cleanly. An AR(2) model is a multiple linear regression on two lag features. With G distinct (store, department) groups, the textbook formulation is G independent regressions. A naive realization issues G concurrent `CREATE MLMODEL` statements (one per group) from a multi-threaded driver. That formulation is correct but a poor fit for an analytic engine: each `CREATE MLMODEL` performs a small fixed amount of work (parse, plan, run a small aggregation, write a catalog entry), and at thousands of groups the per-statement overhead dominates the actual numerical work.

The formulation we describe here, which the architecture supports natively, expresses all G regressions as a *single* relational query that uses the matrix engine described in Section 5. This both eliminates the per-statement overhead and aligns with the paper's central design point: matrix algebra as ordinary SQL.

Implementation. A multiple linear regression has the closed-form solution $\hat{\beta} = (X^T X)^{-1} X^T y$. With Ocient's first-class matrix type and `INVERSE` operator, the solution per group is a single SQL expression. Per-group $X^T X$ and $X^T y$ are scalar SUM aggregations under `GROUP BY (store, department)`; we then assemble

each group's X^TX as a 3×3 matrix and X^Ty as a 3×1 vector via the matrix-literal constructor, multiply by the inverse, and persist the resulting coefficient vector as a column of the output table (Listing 2):

Listing 2: UC3 trained as a single CTE: per-group AR(2) via native matrix INVERSE

```
CREATE TABLE uc3_coefs AS
WITH lagged AS (
  SELECT store, department,
    LAG(weekly_sales,1) OVER (PARTITION BY store, department ORDER
        BY week) AS lag1,
    LAG(weekly_sales,2) OVER (PARTITION BY store, department ORDER
        BY week) AS lag2,
    weekly_sales AS y FROM uc3_stage),
filtered AS (SELECT * FROM lagged WHERE lag1 IS NOT NULL AND lag2
    IS NOT NULL),
sums AS (
  SELECT store, department,
    SUM(lag1*lag1) AS s11, SUM(lag1*lag2) AS s12, SUM(lag1) AS s13,
    SUM(lag2*lag2) AS s22, SUM(lag2) AS s23, CAST(COUNT(*) AS
        DOUBLE) AS s33,
    SUM(lag1*y)   AS sy1, SUM(lag2*y)   AS sy2, SUM(y)   AS sy3
  FROM filtered GROUP BY store, department)
SELECT store, department,
  INVERSE({{s11,s12,s13},{s12,s22,s23},{s13,s23,s33}})
    * {{sy1},{sy2},{sy3}} AS coefs
FROM sums;
```

Serving and scoring are likewise a single CTE: at the most recent observed week, join `uc3_coefs` with the lagged values and compute $\hat{y} = a \cdot \texttt{lag1} + b \cdot \texttt{lag2} + c$ where $(a, b, c)^T$ are the per-group coefficients. The product `coefs` is a 3×1 coefficient matrix per group, indexed in SQL with `coefs[1,1]`, `coefs[2,1]`, `coefs[3,1]`.

Performance. Both formulations solve the same normal equations and therefore produce bit-identical MSLE; the difference is engine cost. In a small-cluster ablation across SF=10 through SF=1,000, the grouped formulation trained the per-group coefficients in 2.7–3.4 s at every scale while the per-key formulation grew from 25 s at SF=10 to over 38 s at SF=100.

7.3 UC4: Statistical Text Classification

UC4 detects spam in product reviews; the metric is Matthews Correlation Coefficient with a target of 0.65. We avoid an external embedding model: a statistical N-gram language model implemented as ordinary SQL aggregations (regex tokenization, array UNNEST, a chain of self-joins for 8-grams, group-by for the per-N-gram spam-odds lookup table) achieves MCC 0.673 at 100 TB. The full pipeline lives in SQL, with no external NLP toolkit and no UDFs.

7.4 UC5: Set-Based Feature Engineering for Price Prediction

UC5 predicts marketplace product prices from free-text descriptions; the metric is RMSLE with target 0.500. We exploit a regularity of price text (Heaps' Law: short phrases such as "leather wallet" are strongly informative and the phrase vocabulary grows sub-linearly with corpus size) to compress the corpus into a "Price Atlas" (a per-N-gram (min, max, avg) of log-price) that a 2-tree, depth-2 GBT consumes. The N-gram construction is the same self-join pattern as UC4; the GBT trains on the small Price Atlas rather than the full corpus, which keeps RMSLE under 0.500 at every scale (0.473 at 100 TB) at training cost that is sub-linear in the data volume (Section 9.2). Listing 3 below shows the N-gram-generation self-join, the only piece of the pipeline that is not a textbook aggregate.

Listing 3: Set-Based N-Gram Generation

```
WITH tokens AS (
  SELECT id, unnest(token_array) WITH ORDINALITY AS (token, idx),
         log_price FROM uc5_token_arrays),
  tokens_and_indexes AS (SELECT id, token, idx - 1 AS minus1 FROM
       tokens)
SELECT t1.id, t1.token || '_' || t2.token AS ngram, t1.log_price
FROM tokens t1 JOIN tokens_and_indexes t2
  ON t1.id = t2.id AND t1.idx = t2.minus1;
```

7.5 UC6: Polynomial Features and Threshold Optimization

UC6 predicts imminent hardware failure from telemetry; the metric is F1 with target 0.190 on a heavily imbalanced positive class. The pipeline is a SQL window function (constructing labeled rows from time-windowed telemetry), a multiple linear regression with polynomial features injected directly in the training SELECT: every base feature x_i raised to powers 1–4 plus all unordered pairwise interactions $x_i \cdot x_j$ $(i < j)$, yielding ≈40 columns total, and a Java-side grid search over the decision threshold using `SUM(CASE ...)` aggregates against the cached score column. F1 exceeds target at small scales but degrades at larger scales due to a property of the synthetic data generator (also observed on Spark, Section 9.3).

7.6 UC7: Product Recommendations via Baseline Predictors

UC7 predicts user-product ratings; the metric is MAE with target 1.800. The textbook formulation is matrix factorization (ALS or SGD over a low-rank user-product matrix), but the simpler Baseline Predictors decomposition (rating ≈ global mean + user bias + product bias) clears MAE 0.729 at 100 TB with only three group-by aggregations and no iteration. The use case illustrates that the relational primitives often suffice to clear the benchmark without resorting to a specialized solver.

7.7 UC8: Pivoted Feature Engineering and JIT Tree Inference

Specification. UC8 classifies shopping trips into trip types from the transaction log. Inputs are line-level scan records (one row per item scanned); the prediction is per-trip (one row per order). The metric is classification accuracy with a target of 0.65.

Why this maps cleanly. The pipeline has two distinct phases that play to different parts of the engine. The feature-engineering phase needs to reshape line-level rows into trip-level feature vectors by pivoting 68 distinct department categories into columns, a textbook large GROUP BY that benefits directly from the bufferless I/O path described in Section 3.1. The serving phase needs to score billions of trips with a deep decision tree (`maxDepth=13`, several thousand internal split nodes), a workload where interpreted-AST evaluation overhead dominates and JIT compilation pays off. UC8 is therefore the use case that exercises both the I/O architecture and the LLVM JIT inference path described in Section 6.

Implementation. The pivot is a single aggregate query with 68 `SUM(CASE ...)` expressions:

Listing 4: Massive Pivot Aggregation

```
SELECT o_order_id,
       SUM(CASE WHEN department = 'FINANCIAL_SERVICES'
           THEN quantity ELSE 0 END) AS financial_services,
       SUM(CASE WHEN department = 'SHOES'
           THEN quantity ELSE 0 END) AS shoes,
       ... (66 more departments) ...
FROM raw_data
GROUP BY o_order_id
```

At scale factor 100,000, this `GROUP BY` triggers a massive shuffle. To optimize the subsequent tree-training queries, the Java driver creates explicit indexes on all 68 generated columns. This adds preprocessing time but significantly accelerates the per-iteration training queries that scan candidate split points.

Sparsity-Aware Workload-Level Binning. Many of the 68 pivoted department columns are sparse: most trips touch only a handful of departments, so a given column is zero for the majority of rows. Splitting on continuous values via the per-node `percent_rank` window costs $O(N \log N)$ on each candidate column at each tree node. To bound that cost, the Java driver computes a sparsity-aware threshold per column in a single aggregate pass over the 71 continuous features (the 68 pivoted department counts plus three base aggregates: `scan_count`, `scan_count_abs`, and `dow`). For columns where zero is the dominant value (frequency $> n/2$), the threshold is the minimum positive value (separating present-from-absent); for the remainder, the threshold is the mean. The driver then materializes a binned 1% sample (`uc8_stage_sample_bin`) in which each continuous column has been mapped to $\{0, 1\}$ against its threshold, builds an index on each binned column, and trains the decision tree (`maxDepth` 13, `numSplits=2`) on the binned sample. At SF=100,000 this collapses UC8 train from a 3,227 s no-binning baseline to 2,045 s (a 37% reduction) without changing accuracy.

Sampled-train, full-inference. Training is on the binned 1% sample; serving binarizes the full pivoted set on-the-fly against the same per-column thresholds (inline `CASE` expressions cross-joined to the persisted thresholds table), so the trained tree always sees $\{0, 1\}$-valued features at both train and serve time. This sampled-train, full-inference pattern is a deliberate choice: tree accuracy plateaus well before the full training set is consumed, so capping training input keeps training time bounded; serving the full pivoted set scores billions of trips through a deep tree, a workload that would dominate without the LLVM JIT path (Section 6) that lowers the per-row interpretation cost.

7.8 UC10: Two-Feature Linear Regression for Fraud Detection

UC10 detects fraudulent transactions; the metric is accuracy with target 0.700. Most of the discriminative signal in the synthetic dataset is captured by two engineered features: time of day, normalized to $[0, 1]$, and transaction amount as a fraction of the account's authorized limit (so > 1.0 already encodes "exceeded limit"). A two-feature multiple linear regression with a 0.5 cutoff produces 0.95 accuracy at every scale factor we tested, in single-digit-second train time even at 100 TB. The pipeline is three queries (a join + projection for the staged features, the `CREATE MLMODEL`, and a thresholded `CASE WHEN` for serving); the engineering point is that a small set of well-chosen features clears the threshold faster and with less complexity than the textbook RF formulation would.

8 SYSTEM OPERATIONS AND GOVERNANCE

Executing queries at 100 TB scale requires robust system-level safeguards that go beyond algorithm correctness. We rely on high-throughput data ingestion and cooperative workload management.

8.1 High-Throughput Data Ingestion

At 100 TB, simply loading the data is a significant challenge. Ocient uses a specialized component called the **Stream Loader**.

Architecture: The Stream Loader is a separate set of nodes dedicated to parsing incoming data streams (from Kafka or S3) and converting them into database pages.

CTAS Integration: For TPCx-AI, when we run a `CREATE TABLE AS SELECT` (e.g., for feature engineering), the query results are piped directly to the Stream Loaders. This lets compute nodes offload persistence work.

8.2 Workload Management (WLM)

To prevent a single massive ML job from destabilizing the cluster, we employ a cooperative scheduling model.

Cooperative Scheduling: Execution is broken down into small units called *operator instances*. Each instance is designed to run for a short time quantum (target ~100 ms, set as a system-level mutable parameter) before yielding control back to the scheduler.

Priority Paging: We do not enforce hard memory quotas per service class. Instead, if the system comes under memory pressure, the WLM pages out data belonging to lower-priority queries to NVMe. This allows ML jobs to burst to use 100% of available RAM if the cluster is otherwise idle, maximizing efficiency.

9 EVALUATION

9.1 Setup

The experiments were conducted on a single Ocient cluster. We use high-end, off-the-shelf enterprise hardware from standard vendors, with no custom silicon and no specialized HPC interconnect, to demonstrate that these results are achievable on parts that any enterprise can purchase through normal channels.

Foundation Nodes (8x): Eight Supermicro AS-1125HS-TNR servers, each equipped with two AMD EPYC 9654 96-Core Processors (total 192 physical cores per node), 2.3 TB of Samsung DDR5 RDIMMs, and 12 Solidigm D7-P5520 NVMe drives. This is 96 NVMe drives across the cluster, providing massive aggregate I/O bandwidth.

SQL Node (1x): One server with the same specification as the foundation nodes, utilized for query parsing, optimization, and ML coordination.

Loader Nodes (2x): Two servers dedicated to data ingestion (ETL).

Network: All servers were interconnected via a 100 Gbps Ethernet fabric using Mellanox ConnectX-6 Dx adapters.

We executed the benchmark at scale factors 1, 10, 100, 1,000, 10,000, and 100,000. Scale factor 1 has a total data volume of 1 GB, while scale factor 100,000 corresponds to an aggregate generated data volume of 100 Terabytes across all defined use cases.

9.2 Scalability Analysis

The detailed performance metrics in Table 1 (visualized in Figure 2) reveal distinct scaling behaviors driven by the algorithmic nature of each use case across the entire 1 GB to 100 TB testing range.

9.2.1 Data Reduction Scaling (UC1, UC5, UC7, UC10). Training time for Price Prediction (UC5) grew only from 22.3 s at 1 GB to 201.9 s at 100 TB despite a 100,000× increase in scale factor. This sub-linear scaling is a direct result of the "Price Atlas" strategy (cf. Listing 3). The cost of generating N-grams scales linearly with the data, but the cost of training the Gradient Boosted Tree depends on the size of the vocabulary (the number of unique N-grams). In many real-world text corpora, the vocabulary size grows according to Heaps' Law (roughly $V \propto n^{\beta}$ where $\beta < 1$) [24]. Use Case 1 (Segmentation), Use Case 7 (Product Recommendations), and Use Case 10 (Fraud Detection) likewise remain remarkably stable across scales: UC1 training stays between roughly 57 and 65 seconds across all six scale factors (56.7–64.7 s), UC7 between 11 and 15 seconds, UC10 between 1 and 9 seconds. This demonstrates the efficiency of Ocient's parallel aggregation operators, which reduce the massive dataset to summary statistics (centroids, average ratings) very early in the pipeline.

9.2.2 Linear Scaling (UC4, UC6, UC8). Workloads dominated by scanning raw text or calculating window functions demonstrated strict linear scaling across the entire spectrum. Decision tree training for Trip Classification (UC8) scaled from 40.8 s at 1 GB up to 2,045 s at 100 TB after applying the workload-level binning optimization described in Section 7.7 (the corresponding pre-binning baseline at 100 TB was 3,227 s). This near-linearity highlights the minimal overhead of the SPDK-based storage engine, ensuring the system processes data proportional to its volume without succumbing to random access latency penalties or OS page thrashing.

9.2.3 Aggregation-Bound Scaling (UC3). Use Case 3 (Sales Forecasting) is the most striking sub-linear case: the grouped CTE formulation (Section 7.2) keeps train time essentially flat between 3.8 and 7.3 seconds across the full 1 GB–100 TB range, despite a 100,000× increase in scale factor. The grouped formulation aggregates all per-(store, department) regression matrices into a single matrix-valued aggregate, so the per-iteration cost depends on the number of distinct groups rather than the number of input rows. The number of distinct (store, department) pairs is bounded by the data generator's parameters and grows much more slowly than the row count; preprocessing scales linearly with the row count but the train phase tracks the group count.

9.3 Model Quality

We met the TPCx-AI model-quality target on seven of the eight executed use cases at every scale factor reported in Table 1; UC6 is the exception, discussed below. UC4 holds MCC above the 0.65 target at every scale (MCC declines from 0.948 at 1 GB to 0.673 at 100 TB; margin above target tightens from 0.298 to 0.023). UC5 stays under the 0.500 RMSLE threshold across the full sweep (0.473 at 100 TB). UC7 holds MAE well below the 1.800 target. UC8 sits at 0.661–0.700 across scales, above the 0.65 target with the margin tightening to 0.011 at 100 TB. UC10 holds 0.95 accuracy independent of scale factor, well above the 0.700 threshold.

For UC6 (Hardware Failure Prediction), our F1 score exceeds the 0.190 target at 1 GB and 10 GB, but degrades at higher scales (0.054 at 1 TB; 0.010 at 100 TB). This is not a property of our implementation: the same degradation appears in our algorithm-matched Spark implementation (Section 9.4) at every measured scale (Spark F1 0.122 at 100 GB, 0.055 at 1 TB, 0.024 at 10 TB, 0.010 at 100 TB; Ocient F1 within rounding), and is intrinsic to the synthetic data generator (the rate of imminent-failure events does not scale proportionally with the rest of the dataset, so the positive class becomes vanishingly rare). We report the numbers as-observed rather than masking them; both engines agree on the degradation pattern, which is the correctness signal we can defend.

9.4 Algorithm-Matched Comparison Against Apache Spark

We wrote algorithm-matched Spark implementations of all eight executed use cases to isolate engine from algorithm choice. Each Spark implementation uses the same algorithm we used in Ocient: closed-form normal equations for UC3, UC6, UC10; Baseline Predictors for UC7; N-gram Price Atlas for UC5; 1% sampled training with full-set inference for UC8; K-Means with 10 Lloyd iterations for UC1; an N-gram log-odds classifier for UC4.

Setup. The Spark cluster is five Supermicro AS-1125HS-TNR servers of the same generation as the Ocient Foundation cluster (192-core EPYC 9654, 2.2 TB DDR5, 7.3 TB local NVMe, 100 Gbps NICs); Spark 3.5.7 runs on YARN with HDFS replication factor 1. The 5-node Spark cluster has 5/8 of the Ocient Foundation node count. The PySpark drivers (~1500 lines) use `pyspark.ml` where directly equivalent and `mapPartitions` with NumPy least-squares for the per-key autoregressive models in UC3. Hyperparameters are held constant across engines with no per-engine tuning: KMeans k=4 with 10 Lloyd iterations, identical 1% UC8 sample fractions, identical N-gram cutoffs, identical tree depth limits.

Results at SF=100 and SF=1,000 (small-scale sanity). At the smallest two scales (100 GB, 1 TB) Spark wins cumulative wall-clock by 1.47× and 1.37× respectively; Ocient already wins UC3 and UC10 while Spark leads on tree-based UC5/UC8 by 2–3×. Algorithm-fidelity holds at every scale (UC4 MCC, UC6 F1, UC8 accuracy from SF=1,000 onward, UC10 accuracy reproduce to within rounding; UC7 carries a ~0.02 Ocient-favored MAE offset; UC3 MSLE and UC5 RMSLE differ for driver-side data-prep and tree-shape reasons respectively). Per-UC tables at SF=100 and SF=1,000 are in the artifacts repository [3]; we report 10 TB and 100 TB here because they carry the architectural story.

Results at SF=10,000 (10 TB). The 10 TB sweep runs in 22.8 minutes on Spark vs 24.8 minutes on Ocient (Spark 9% ahead overall), with Ocient already winning UC1, UC3, UC4, and UC10 outright;

Table 1: Detailed Performance and Quality Metrics for TPCx-AI Use Cases on Ocient across scale factors. All times are in seconds. UC: Use Case, MAE: Mean Absolute Error, RMSLE: Root Mean Squared Log Error.

Use Case	Data Volume	Preproc. (s)	Train (s)	Serve (s)	Metric	Target	Lower Better	Score
Segmentation (UC1)	1 GB	15.1	57.7	6.9	N/A	N/A	N/A	N/A
	10 GB	7.6	57.4	3.4	N/A	N/A	N/A	N/A
	100 GB	8.9	56.7	4.4	N/A	N/A	N/A	N/A
	1 TB	13.3	61.4	4.8	N/A	N/A	N/A	N/A
	10 TB	35.3	64.0	4.4	N/A	N/A	N/A	N/A
	100 TB	117.5	64.7	5.6	N/A	N/A	N/A	N/A
Forecasting (UC3)	1 GB	11.3	4.2	4.2	MSLE	5.400	Yes	0.092
	10 GB	6.6	4.2	5.3	MSLE	5.400	Yes	0.058
	100 GB	6.4	3.8	3.6	MSLE	5.400	Yes	0.028
	1 TB	10.8	7.3	4.2	MSLE	5.400	Yes	0.044
	10 TB	25.3	4.2	4.3	MSLE	5.400	Yes	0.047
	100 TB	77.0	4.1	4.2	MSLE	5.400	Yes	0.058
Spam Detect. (UC4)	1 GB	17.2	6.6	5.3	MCC	0.650	No	0.948
	10 GB	17.1	4.2	6.0	MCC	0.650	No	0.862
	100 GB	30.4	7.9	7.2	MCC	0.650	No	0.755
	1 TB	49.2	14.4	8.2	MCC	0.650	No	0.718
	10 TB	115.6	29.4	17.6	MCC	0.650	No	0.696
	100 TB	398.0	93.3	41.2	MCC	0.650	No	0.673
Price Pred. (UC5)	1 GB	33.3	22.3	4.6	RMSLE	0.500	Yes	0.486
	10 GB	33.3	27.7	11.7	RMSLE	0.500	Yes	0.479
	100 GB	35.5	30.0	6.5	RMSLE	0.500	Yes	0.476
	1 TB	38.1	43.0	3.9	RMSLE	0.500	Yes	0.474
	10 TB	46.4	90.0	6.8	RMSLE	0.500	Yes	0.473
	100 TB	61.3	201.9	5.0	RMSLE	0.500	Yes	0.473
HW Failure (UC6)	1 GB	10.6	16.5	11.8	F1 Score	0.190	No	0.452
	10 GB	9.2	15.7	12.6	F1 Score	0.190	No	0.249
	100 GB	8.8	16.0	11.5	F1 Score	0.190	No	0.120
	1 TB	14.2	18.6	17.9	F1 Score	0.190	No	0.054
	10 TB	52.6	58.7	30.9	F1 Score	0.190	No	0.025
	100 TB	294.6	239.2	151.4	F1 Score	0.190	No	0.010
Recommend. (UC7)	1 GB	3.2	11.5	6.3	MAE	1.800	Yes	0.710
	10 GB	4.4	14.6	7.4	MAE	1.800	Yes	0.715
	100 GB	3.9	13.7	7.4	MAE	1.800	Yes	0.720
	1 TB	5.8	13.7	6.4	MAE	1.800	Yes	0.723
	10 TB	8.0	14.8	7.2	MAE	1.800	Yes	0.726
	100 TB	11.9	12.3	7.4	MAE	1.800	Yes	0.729
Trip Class. (UC8)	1 GB	31.4	40.8	5.3	Accuracy	0.650	No	0.700
	10 GB	56.3	41.4	4.1	Accuracy	0.650	No	0.678
	100 GB	59.3	79.3	4.3	Accuracy	0.650	No	0.662
	1 TB	107.3	184.9	11.2	Accuracy	0.650	No	0.667
	10 TB	225.7	572.6	19.1	Accuracy	0.650	No	0.664
	100 TB	668.7	2044.9	38.0	Accuracy	0.650	No	0.661
Fraud Detect. (UC10)	1 GB	3.6	2.2	3.7	Accuracy	0.700	No	0.949
	10 GB	8.1	1.1	4.7	Accuracy	0.700	No	0.951
	100 GB	6.7	1.9	9.1	Accuracy	0.700	No	0.949
	1 TB	12.7	2.1	12.6	Accuracy	0.700	No	0.950
	10 TB	26.2	4.1	27.4	Accuracy	0.700	No	0.950
	100 TB	59.2	8.8	90.5	Accuracy	0.700	No	0.950

the 4–4 per-UC split at 10 TB foreshadows the same split at 100 TB. The per-UC table at SF=10,000 is in the artifacts repository [3].

Results at SF=100,000 (100 TB). We staged to HDFS at replication factor 1 and ran the full sweep at the same scale Ocient executes natively; all eight use cases ran to completion on Spark (Table 2).

Ocient wins four use cases by large factors (UC1 188 s vs. 1942 s, 10.3×, the join-heavy preprocess that Spark's HDFS-block reads cannot close against bufferless I/O; UC3 85 s vs. 410 s, 4.8×, with the grouped formulation described in Section 7.2 replacing the per-key path; UC4 533 s vs. 835 s, 1.6×; UC10 159 s vs. 654 s, 4.1×). Spark wins the remaining four, three by smaller factors (UC6 638 s vs. 685 s, 1.07×, with both engines failing the F1 target; UC7 23 s vs. 32 s, 1.4×; UC5 72 s vs. 268 s, 3.7×) and one substantially (UC8 1,089 s vs. 2,752 s, 2.5×). UC5 and UC8 are the tree-based cases: both stem from the per-node `percent_rank` split-scan path used by the regression-tree and decision-tree implementations; the histogram-binning gap with Spark MLlib is the principal architectural delta on tree-based models. Despite the 4–4 per-UC split, Ocient's wins are large enough in magnitude that the raw cumulative total favors Ocient by 17% (4,701 s vs. 5,663 s). The equal-hardware projection below normalizes for the 5-vs-8 node count.

Trend across scales and net result. Three trends emerge from Table 2 (per-UC at the headline scale) and Table 3 (cumulative). On data-scan-bound use cases (UC1, UC3, UC4, UC10), Ocient's

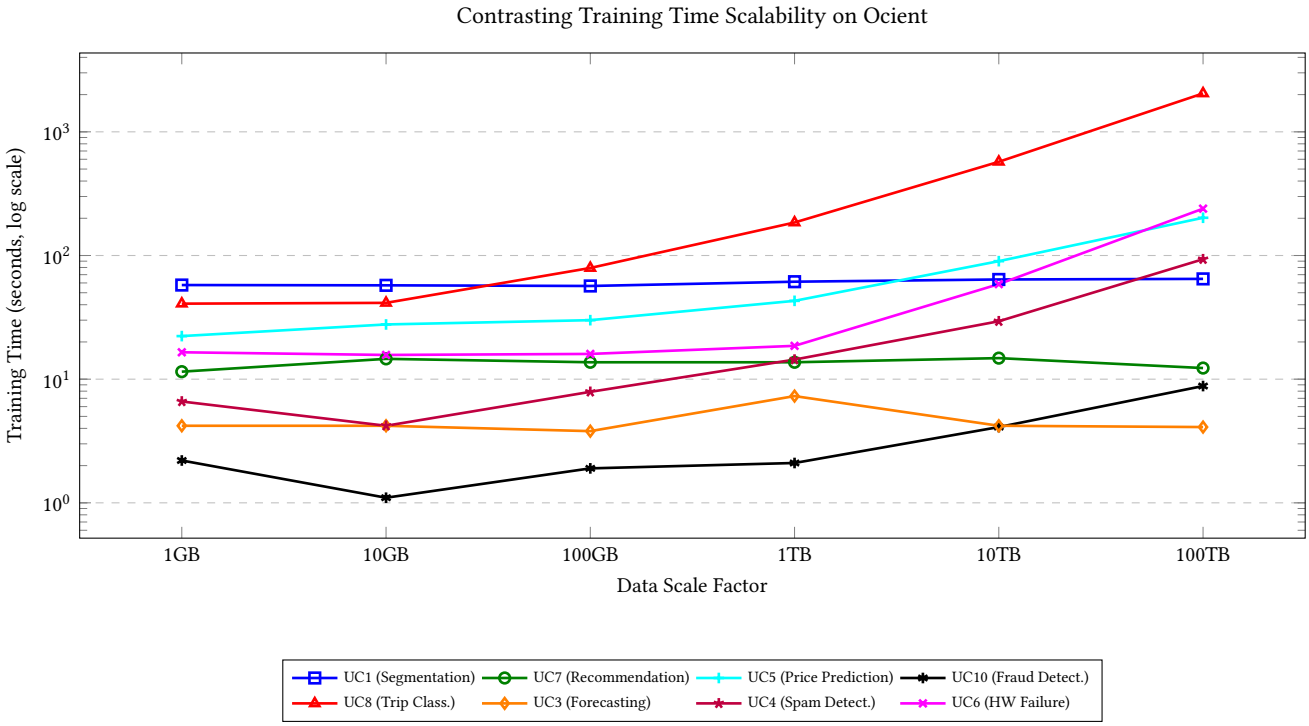

Figure 2: Training time scalability for the eight evaluated TPCx-AI use cases on a log-log scale.

Table 2: Algorithm-matched Spark vs Ocient at SF=100,000 (100 TB). Times in seconds. Ocient uses the grouped UC3 formulation throughout (Section 7.2); per-UC accuracy metrics are reported in Section 9.3 and the artifacts repository.

UC	Spark (5 nodes)			Ocient (8 nodes)		
	pre	train	serve	pre	train	serve
UC1	1929.4	11.8	1.1	117.5	64.7	5.6
UC3	406.3	2.8	0.4	77.0	4.1	4.2
UC4	617.2	65.2	152.7	398.0	93.3	41.2
UC5	59.9	10.4	2.1	61.3	201.9	5.0
UC6	380.3	211.1	46.5	294.6	239.2	151.4
UC7	12.8	4.6	5.9	11.9	12.3	7.4
UC8	955.1	55.8	78.0	668.7	2044.9	38.0
UC10	416.9	173.3	63.3	59.2	8.8	90.5

per-UC advantage *widens monotonically with scale*: at SF=100,000, UC1 is 10.3× Ocient, UC3 is 4.8×, UC10 is 4.1×, and UC4 narrows from a 1.07× Spark win at SF=100 to a 1.6× Ocient win. UC5 and UC8 (tree-based) are Spark-favored at every scale (3.7×, 2.5× at SF=100,000); Spark MLlib's partition-local histogram binning beats our per-node `percent_rank` split-scan, the architectural change in Section 11. Algorithm-fidelity is preserved at every scale (UC4 MCC, UC10 accuracy, UC6 F1 reproduce to within rounding on both engines).

Equal-hardware extrapolation. The 5-node Spark cluster vs. 8-node Ocient Foundation comparison is asymmetric. To give a fair reading we report a node-normalized projection alongside the raw numbers: Spark times multiplied by $5/8 = 0.625$ (perfect-linear scaling, the most optimistic assumption for Spark; realistic 5→8 scaling is sub-linear, so the normalized position is conservative for Ocient). Table 3 reports the cumulative comparison at every scale. Raw at SF=100,000 favors Ocient by 1.20× (4,701 s vs. 5,663 s); the equal-hardware projection favors Spark by 1.33× (vs. 3,539 s).

Table 3: Cumulative wall-clock comparison: raw 5n Spark vs. 8n Ocient, and equal-hardware projection (Spark ×5/8, optimistic linear scaling). Spark times in seconds; the projection is conservative for Ocient.

Scale	Spark 5n raw	Spark → 8n projected	Ocient 8n raw	Cum. ratio Oc/Sp@8n
SF=100 (100 GB)	287.5	179.7	423.2	Spark 2.36×
SF=1,000 (1 TB)	488.1	305.1	666.0	Spark 2.18×
SF=10,000 (10 TB)	1,367.9	854.9	1,490.6	Spark 1.74×
SF=100,000 (100 TB)	5,662.9	3,539.3	4,700.7	Spark 1.33×

Projected crossover. The Ocient-to-8-node-Spark cumulative ratio narrows monotonically with scale (Table 3). A least-squares fit of cumulative wall-clock against scale factor across the four measured points favors a quadratic model $y = a \cdot \mathrm{SF}^2 + b \cdot \mathrm{SF} + c$ (combined $R^2 = 0.9994$) that projects the equal-hardware crossover at SF ≈ 136,000, just past our largest measured scale. We do not claim a measured crossover at any scale we executed; measuring cumulative wall-clock at SF=150,000 and SF=200,000 to validate the projection is on our future-work list (Section 11).

Per-UC structure at SF=100,000 under equal-hardware projection. Ocient wins UC1 (6.5×), UC3 (3.0×), UC10 (2.6×), and ties UC4 (1.02× Spark); Spark wins UC5 (5.9×), UC8 (4.0×), UC7 (2.2×), UC6 (1.7×). The Ocient-favored set is the data-scan-bound and per-key-regression cases the architecture was designed for; the Spark-favored set is tree-training (UC5, UC8) where the architectural delta is the histogram-binning gap (Section 11).

10 RELATED WORK

ML-in-database has a rich history [22]; we position our work along three axes prior systems have explored separately: *algorithm exposure*, *execution path*, and *data path*.

Three prior approaches. **MADlib** [7, 14] and **Bismarck** [11] expose ML as C UDFs with state in user-defined aggregates: fast math, opaque to the planner. **SystemML** [6] keeps optimizer visibility but lowers a separate DSL (DML) to a separate Spark/MR runtime. **Vertica** [18, 29] and **Microsoft SQL Server** [27] embed R/Python; **Google BigQuery ML** [13] and **Amazon Redshift ML** [1] ship rows to a hosted training service; both incur data-motion costs that grow with scale.

Where Ocient Differs. The IRLS update in Section 5.1 is the same Newton-Raphson iteration MADlib implements as a UDF and SystemML lowers from a DML script; the difference is where the iteration sits. In our system it is an ordinary SQL CTE: `TRANSPOSE`/`SUM`/`INVERSE` are operators in the same planner as the rest of the query, the per-iteration scan reuses the SPDK I/O path, and per-row Hessian contributions land in the same hugepage-fragment buffer aggregation uses. **HyPer** [33] demonstrated planner-integrated in-situ ML in a main-memory engine; we extend that approach to disk-resident shared-nothing storage at 100 TB, with Section 9.4 quantifying where the integration pays back at scale.

11 FUTURE WORK

Four directions are active. *Empirical crossover validation:* measure SF=150,000 and SF=200,000 on both engines to land the projected equal-hardware crossover (Section 9.4). *Native partition-local histogram binning:* close the residual UC5/UC8 tree-training gap (workload-level binning already recovered 37% of UC8 train at 100 TB). *Deep-learning layer types* for UC2/UC9 (Section 2.2). *Mid-query fault tolerance* via sub-query DAG lineage replay for stricter-availability deployments.

12 CONCLUSION

We presented the first unofficial published TPCx-AI results at 100 TB, executing eight of ten use cases natively in a relational engine via six architectural pillars (Sections 3–6). The algorithm-matched Spark comparison (Section 9.4) reports raw and equal-hardware-projected cumulative wall-clock; under the equal-hardware projection the Ocient-to-8-node-Spark cumulative ratio narrows monotonically with scale and a quadratic fit projects a crossover just past SF=100,000. The persistent Spark wins on UC5/UC8 trace to a histogram-binning gap rather than the storage architecture; load-bearing accuracy targets reproduce on both engines to within rounding at every scale.

REFERENCES

[1] Amazon Web Services. 2026. Amazon Redshift ML. Retrieved June 30, 2026 from https://aws.amazon.com/redshift/features/redshift-ml/

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2–11. https://doi.org/10.1109/SUPERC.1990.129995

[3] Jason Arnold, Neesh Dahiya, and Knut Stolze. 2026. Companion Artifacts for Benchmarking Native In-Database TPCx-AI at 100 Terabytes in Ocient: SQL Templates, Java Driver, and Spark Reference. Retrieved June 30, 2026 from https://github.com/Ocient/vldb2026-tpcxai

[4] Jens Axboe. 2019. Efficient IO with `io_uring`. Retrieved June 30, 2026 from https://kernel.dk/io_uring.pdf Kernel.org documentation.

[5] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqi, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research (CIDR 2020)*. www.cidrdb.org, Amsterdam, The Netherlands, 1–8. Retrieved June 30, 2026 from http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[6] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Engineering Bulletin* 37, 3 (2014), 52–62. Retrieved June 30, 2026 from http://sites.computer.org/debull/A14sept/p52.pdf

[7] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1481–1492. https://doi.org/10.14778/1687553.1687576

[8] Chris Currier. 2022. Protocol Buffers. In *Mobile Forensics - The File Format Handbook - Common File Formats and File Systems Used in Mobile Devices*, Christian Hummert and Dirk Pawlaszczyk (Eds.). Springer, Cham, Switzerland, 223–260. https://doi.org/10.1007/978-3-030-98467-0_9

[9] Dell Technologies. 2023. Dell Reinforces its TPCx-AI Benchmark Leadership using the 16G PowerEdge R6625. Retrieved June 30, 2026 from https://infohub.delltechnologies.com/en-us/p/dell-reinforces-its-tpcx-ai-benchmark-leadership-using-the-16g-poweredge-r6625-hardware-platform-at-sf1000/

[10] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 44–62. https://doi.org/10.1109/69.50905

[11] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, Scottsdale, AZ, USA, 325–336. https://doi.org/10.1145/2213836.2213874

[12] Gene H. Golub and Charles F. van Loan. 2013. *Matrix Computations* (fourth ed.). The Johns Hopkins University Press, Baltimore, MD, USA.

[13] Google Cloud. 2026. Introduction to ML in BigQuery. Retrieved June 30, 2026 from https://cloud.google.com/bigquery/docs/bqml-introduction

[14] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: or MAD Skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[15] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/NECO.1997.9.8.1735

[17] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). ICLR, San Diego, CA, USA, 1–15. Retrieved June 30, 2026 from http://arxiv.org/abs/1412.6980

[18] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[19] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, Washington, DC, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323. https://doi.org/10.1145/355841.355847

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[22] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine Learning for Databases. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3190–3193. https://doi.org/10.14778/3476311.3476405

[23] libuv contributors. 2026. libuv: Cross-platform asynchronous I/O. Retrieved June 30, 2026 from https://github.com/libuv/libuv

[24] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511809071

[25] Donald W. Marquardt. 1963. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *J. Soc. Indust. Appl. Math.* 11, 2 (1963), 431–441. https://doi.org/10.1137/0111030

[26] Pedro Martins, Paulo Tomé, Cristina Wanzeller, Filipe Sá, and Maryam Abbasi. 2021. Comparing Oracle and PostgreSQL, Performance and Optimization. In *Trends and Applications in Information Systems and Technologies*, Álvaro Rocha, Hojjat Adeli, Gintautas Dzemyda, Fernando Moreira, and Ana Maria Ramalho Correia (Eds.). Springer International Publishing, Cham, 481–490.

[27] Microsoft Corporation. 2026. SQL Machine Learning documentation. Retrieved June 30, 2026 from https://learn.microsoft.com/en-us/sql/machine-learning/

[28] Jorge Nocedal and Stephen J. Wright. 1999. *Numerical Optimization*. Springer, New York, NY, USA. https://doi.org/10.1007/B98874

[29] Carlos Ordonez and Sasi K. Pitchaimalai. 2010. Bayesian Classifiers Programmed in SQL. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2010), 139–144. https://doi.org/10.1109/TKDE.2009.127

[30] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, et al. 2020. MLPerf Inference Benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA 2020)*. IEEE, Valencia, Spain, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[31] Richard D. Schlichting and Fred B. Schneider. 1983. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.* 1, 3 (Aug. 1983), 222–238. https://doi.org/10.1145/357369.357371

[32] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 815–823. https://doi.org/10.1109/CVPR.2015.7298682

[33] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019) (Lecture Notes in Informatics (LNI))*, Vol. P-289. Gesellschaft für Informatik, Bonn, Bonn, Germany, 247–266. https://doi.org/10.18420/btw2019-16

[34] Knut Stolze, Andrew Park, and Jason Arnold. 2025. Beyond Big Data – The Ocient Hyperscale Data Warehouse. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025) (Lecture Notes in Informatics (LNI))*, Vol. P-361. Gesellschaft für Informatik e.V., Bonn, Germany, 679–690. https://doi.org/10.18420/BTW2025-34

[35] Michael Stonebraker. 1986. The Case for Shared Nothing. *IEEE Database Engineering Bulletin* 9, 1 (1986), 4–9.

[36] The SPDK Community. 2025. Storage Performance Development Kit (SPDK). Retrieved June 30, 2026 from https://spdk.io/

[37] Transaction Processing Performance Council. 2023. TPC Newsletter 2023 Q2. Retrieved June 30, 2026 from https://tpc.org/newsletter/tpc-newsletter-2023-07-31.pdf

[38] Transaction Processing Performance Council. 2024. TPC Benchmark™ x-AI (TPCx-AI) Standard Specification Version 2.0.0. Retrieved June 30, 2026 from http://www.tpc.org/tpcx-ai/

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc., 5998–6008. Retrieved June 30, 2026 from https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[40] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995), 20–24. https://doi.org/10.1145/216585.216588

[41] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, Alessandro Warth (Ed.). ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. USENIX Association, Boston, MA, USA, 1–7. Retrieved June 30, 2026 from https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets