Virtualizing Recursion: Just-In-Time Graph Analytics in a Hyperscale Relational Warehouse

Jason Arnold
Ocient
Chicago, Illinois, USA
jarnold@ocient.com

Knut Stolze
Ocient
Jena, Germany
stolze@ocient.com

ABSTRACT

Modern hyperscale data warehouses (MPP) face increasing pressure to support complex, non-linear query paradigms. While the industry is moving toward standardized graph query languages like ISO GQL and SQL:2023 (SQL/PGQ), supporting these on shared-nothing architectures traditionally requires specialized, heavyweight physical operators (e.g., recursive union). This complicates the scheduler and often forces a trade-off between latency and scale. Furthermore, in high-availability environments where metadata operations require distributed consensus, the latency of creating ephemeral tables for recursion becomes prohibitive.

This paper presents Ocient's solution: decoupling the *definition* of recursion from the *execution* via Dataflows: a procedural superset of SQL. This architecture enables support for recursive SQL and is well-positioned as a transpilation target for emerging GQL and SQL:2023 workloads. We describe the implementation of a table virtualization runtime that inspects intermediate row counts to automatically switch between in-memory CTE injection (for low latency) and NVMe-backed storage (for massive scale). The system transparently transitions storage-backed tables back into virtual memory objects as they shrink or vice versa on growing tables. This unified substrate allows Ocient to execute hyperscale graph algorithms (such as Weakly Connected Components on graphs exceeding 100 billion edges) directly within the relational warehouse.

PVLDB Reference Format:
Jason Arnold and Knut Stolze. Virtualizing Recursion: Just-In-Time Graph Analytics in a Hyperscale Relational Warehouse. PVLDB, 19(12): 4103 - 4115, 2026.
doi:10.14778/3827998.3828019

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

1 INTRODUCTION

In the domain of massively parallel processing (MPP) databases, the primary design goal is typically throughput for large scans and joins, tracing back to the seminal Gamma project [10]. However, enterprise workloads increasingly demand the ability to query graph-like structures (such as supply chain bills of materials, financial fraud networks, and identity resolution graphs) directly within the data warehouse.

The SQL standard addresses this via the *WITH RECURSIVE* clause [17]. While new standards like GQL [19] and SQL:2023 (SQL/PGQ) [18] are emerging to handle complex pattern matching, implementing any form of recursion in a shared-nothing, distributed architecture presents a dilemma:

(1) **Client-Side Iteration:** The application fetches a dataset, finds the next hop, and queries again. This introduces massive network latency and serialization overhead (the N+1 query problem [39]).
(2) **Kernel Integration:** The database engine implements specific physical operators (e.g., recursive union and other specialized operators) that handle recursion state. In an MPP system, such operators are notoriously difficult to schedule, as they introduce cycles into the query pipeline (DAG). A cycle in the execution graph complicates pipeline management, deadlock detection, and memory management for spilling large intermediate states [20].

Furthermore, strict adherence to the SQL standard for recursion introduces expressivity gaps. Standard SQL recursion must be monotonic (additive only) and is typically restricted to linear recursion [24]. This makes efficient algorithms like Bidirectional Search [15] or Stable Marriage [38] challenging to express in a single query.

Ocient chose a different path. We observed that the imperative steps required for recursion (iteration, variable state, and conditional branching) offer a unique opportunity for just-in-time (JIT) optimization. We implemented **Dataflows**: a procedural superset of SQL that executes directly on the database. Unlike typical stored procedures, Dataflows are ephemeral and anonymous.

The Dataflow Runtime acts as an intelligent orchestrator. By inspecting the intermediate state of the recursion between iterations, the runtime can make dynamic decisions that a static query optimizer cannot. This enables optimizations such as Table Virtualization, where temporary tables are kept in memory and injected as query literals, and Adaptive Re-planning. Both are applied dynamically during execution.

1.1 Example: k-Hop Reachability

A user wants every vertex reachable from a given source within k hops. In standard SQL this is the canonical recursive common table expression (CTE): an anchor selecting the source, a recursive member that joins the working set against an edge table, and *UNION ALL* accumulating reached vertices. The user submits exactly that:

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. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828019

```
WITH RECURSIVE Reachable(node_id, depth) AS (
    SELECT 42 /* @start_node */, 0
  UNION ALL
    SELECT e.dest, r.depth + 1
    FROM Reachable r JOIN Edges e ON r.node_id = e.src
    WHERE r.depth < 100 /* @max_depth */
)
SELECT node_id, MIN(depth) AS depth FROM Reachable GROUP
     BY node_id;
```

Listing 1: The user's input: a standard recursive CTE

In Ocient, the recursion is detected after parsing (cf. Section 3.3) and the transpiler (cf. Section 3.2) rewrites the CTE into the imperative Dataflow form below, using a frontier table *#Q* for the working set and an accumulator table *#R* for the result:

```
BEGIN QUERY DATAFLOW
  DECLARE @start_node BIGINT = 42;
  DECLARE @max_depth BIGINT = 100;
  CREATE TABLE #R AS SELECT @start_node AS node_id, 0 AS
       depth;
  CREATE OR REPLACE TABLE #Q AS SELECT * FROM #R;
  WHILE ((SELECT count(*) FROM #Q) > 0) DO
    CREATE OR REPLACE TABLE #N AS
      SELECT e.dest AS node_id, q.depth + 1 AS depth
      FROM #Q q JOIN Edges e ON q.node_id = e.src
      WHERE q.depth < @max_depth;
    INSERT INTO #R SELECT * FROM #N;
    DROP TABLE #Q;
    ALTER TABLE #N RENAME TO #Q;
  END WHILE;
  RETURN SELECT node_id, MIN(depth) AS depth FROM #R
       GROUP BY node_id;
END DATAFLOW;
```

Listing 2: The transpiled imperative Dataflow

A typical execution traces the entire architecture: the very first iteration's frontier holds a single row, so the runtime virtualizes *#Q* on the SQL Node and injects it as a literal CTE into the join (cf. Section 4.1). After a few hops the frontier crosses the configurable threshold for the number of rows and the runtime materializes *#Q* to NVMe, switching the planner from broadcast joins to shuffle joins (cf. Section 4.5). Each iteration releases its Workload Manager (WLM) slot before the next loop check (cf. Section 5.1), and the result query *SELECT node_id, MIN(depth) ... FROM #R* is hot-swapped onto the original client connection so a JDBC/ODBC caller sees only a normal result set (cf. Section 3.3).

We use various examples throughout the paper to highlight different aspects. The case studies in Section 8 generalize it: Bidirectional Search runs two coupled BFS frontiers; Weakly Connected Components turns it into a label-propagation BFS over the entire graph; Stable Marriage layers non-monotonic state on top.

1.2 Paper Organization

Section 2 details the Ocient architecture. Section 3 introduces the Dataflow language and how recursive CTEs are handled. Sections 4–6 cover the JIT runtime, operational safety, and storage; Section 7 positions our approach architecturally. Finally, Sections 8 and 9 present algorithmic case studies and experimental evaluations, followed by related work in Section 10. We conclude and provide an outlook in Section 11.

2 BACKGROUND: OCIENT ARCHITECTURE

The Ocient Hyperscale Data Warehouse (Ocient) is a massively parallel processing (MPP) system designed to efficiently store and analyze petabyte-scale datasets [36]. Understanding the underlying hardware-software co-design is essential to understanding why the Dataflow approach is efficient.

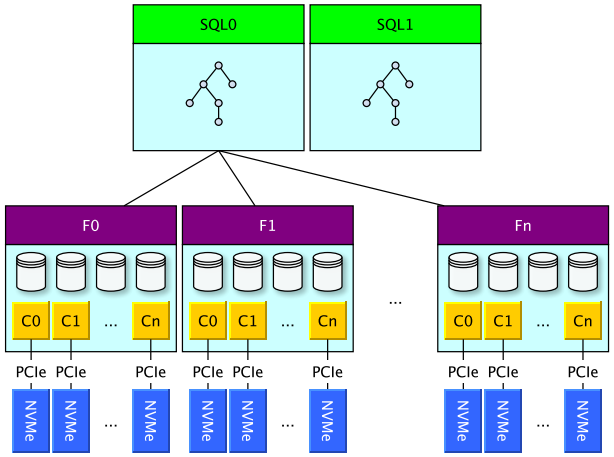

Figure 1: Ocient Architecture: SQL Nodes (e.g., SQL0 and SQL1) manage control flow while Foundation Nodes (F0, ..., Fn) execute heavy compute adjacent to NVMe storage.

2.1 Compute-Adjacent Storage (CASA)

Ocient utilizes a Compute-Adjacent Storage Architecture (CASA), where storage and compute resources are co-located to minimize data movement [36]. The system is deployed on commodity hardware with NVMe drives directly attached to dedicated CPU cores via PCIe lanes.

Ocient does not utilize a traditional buffer pool. Given the extreme IOPS provided by NVMe and SPDK (Storage Performance Development Kit) [14], data is read directly from storage into execution memory, similar to modern *lean storage* architectures proposed in academia [23]. This architectural choice simplifies recursion: intermediate states (temporary tables) are written to NVMe and read back at line-rate, preventing the memory pressure and buffer pool thrashing common in recursive workloads. When a recursive frontier grows larger than main memory, the system does not need to enter a slow spill-to-disk exception path; reading from disk is the standard execution path.

2.2 Distributed Query Execution

Query processing is split between **SQL Nodes** (Coordinators) and **Foundation Nodes** (Compute) [36].

- **SQL Nodes:** Manage the query plan and control flow.
- **Foundation Nodes:** Perform the heavy lifting (Scan, Filter, Join). They operate using a push-based execution model with lock-free Single-Producer Single-Consumer (SPSC) queues between operators for maximum parallelism.

The Dataflow architecture leverages this split: the control logic (the *WHILE* loop) resides on the SQL Node, while the massive data processing (the *JOIN* inside the loop) is pushed down to the Foundation Nodes. Because the Foundation Nodes process data at NVMe speeds, the latency of the Coordinator loop is amortized over the heavy compute time.

3 THE DATAFLOW: A PROCEDURAL SQL SUPERSET

To enable recursion, Ocient introduced the Dataflow. While originally conceived as an execution substrate for recursion, we recognized that providing a general-purpose procedural extension to standard SQL would solve a broader class of problems. We designed Dataflows to be a user-facing language that allows developers to write logic that is historically difficult to express in declarative SQL.

A single recursive step is handled as an iteration of the *WHILE* loop. State between the loop iterations is managed via dataflow-scoped temporary tables.

3.1 Language Semantics

A Dataflow acts as an anonymous, imperative script. It allows a client to send a batch of logic that is parsed, compiled, and executed in a single request.

Key features include:

- **Flow Control:** *IF/THEN/ELSE/ELSEIF*, *WHILE* loops with optional *MAX ITERATIONS* clauses, and both C-style and *FOR ... IN* loops, plus *BREAK*, *CONTINUE*, and *RAISE* for explicit termination and error signalling.
- **Variables:** Strongly typed variables (e.g., *DECLARE @depth INT*) with a *@* prefix, optional initializers, and *NOT NULL* constraints, used for loop termination, scalar accumulation, and intermediate counts.
- **Scoped Temp Tables:** Tables named with a *#* prefix (e.g., *#Queue*) are local to the Dataflow execution. These tables are the primary target for our JIT virtualization optimizations (cf. Section 4).
- **Embedded SQL Statements:** The body admits Ocient's DDL surface (*CREATE / ALTER / DROP TABLE | VIEW | INDEX*, *CREATE OR REPLACE TABLE AS SELECT*), the bulk DML statements relevant to recursion (*INSERT*, *DELETE*, *TRUNCATE*; *UPDATE* is intentionally not supported, as Ocient is append-only), and *CALL* for stored-procedure invocation. Any *SELECT* that runs as a stand-alone statement is also valid as a Dataflow expression source.
- **Exception Handling:** A *FINALLY* block ensures cleanup actions occur regardless of success or failure.

A key distinction between Ocient Dataflows and traditional Stored Procedures is their ad-hoc nature. Stored Procedures typically require administrative rights to create, are persistent catalog objects, and require clients to use specialized protocols to invoke. Dataflows are purely ephemeral: they are sent as standard SQL strings, executed immediately, and vanish upon completion. While Dataflows expose imperative control to the user, the architectural motivation for this design choice (rather than hiding adaptivity inside a declarative executor) is detailed in Section 7.4.

3.2 Rewriting *WITH RECURSIVE*

The transpiler rewrites declarative *WITH RECURSIVE* CTEs into imperative Dataflow scripts via Source Slicing: it consults the query AST to find offsets for the recursive name and member, then performs the rewrite directly on the source string and re-parses the result. This text-rewriting approach trades a small re-parse cost for a robust path that hands the rewritten statement straight to the engine's standard execution paths. Seminaive evaluation [3] drives the loop body: the recursive member writes to *#Next* on every iteration; rows are appended to a *#Result* accumulator and *#Next* is renamed to *#Queue* for the next pass. For *UNION* (set semantics) the transpiler emits an additional *EXCEPT* stage against *#Result* so the optimizer can parallelize deduplication via hash-based aggregation and anti-joins, rather than rely on a single-threaded "seen set." Recursive views are handled by Source Linearization: the validator appends the view's source text to the query buffer and shifts the view's AST node offsets to point to the appended copy, after which the standard transpiler rewrites both layers as one continuous string. The rewrite operates on post-validation AST offsets, so name resolution, scoping, and view expansion are inherited from the validator rather than re-implemented. (Section 1.1 shows the canonical input/output.)

3.3 Transparent Connection State Management (The "Hot Swap")

A unique aspect of our implementation is how the Dataflow is invoked in a way that is transparent to the client driver. The client sends a standard SQL query containing recursion; the server must execute a complex script, yet eventually return a standard result set. This is handled by a "Hot Swap" mechanism.

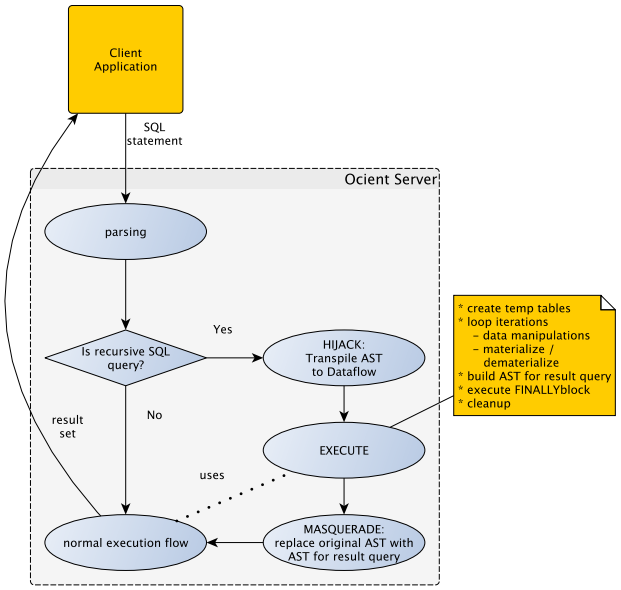

Figure 2: The Connection "Hot Swap" enables backward compatibility with standard JDBC/ODBC clients.

3.3.1 The Hijack-Execute-Masquerade Cycle. The connection logic proceeds in four distinct phases that are invisible to the client:

(1) **Hijack Phase:** Upon detecting recursion, the connection stops processing the original AST. It compiles the Dataflow script and swaps it into the execution buffer.
(2) **Script Execution Phase:** The connection instantiates a runtime, which runs the entire imperative loop (creating tables, inserting data, iterating). Crucially, this is a blocking operation from the perspective of the connection thread.
(3) **Result Retrieval Phase:** Once the Dataflow hits the *RETURN* statement, the processor extracts the expanded result query (e.g., *SELECT * FROM #Result*). It does not execute it yet; it returns the string to the connection.
(4) **Masquerade Phase:** The connection takes this result query string, parses it into a new AST, and replaces its own internal state. It registers a cleanup callback to run the *FINALLY* block later. It then proceeds to validation and execution as if the client had originally sent the simple *SELECT * FROM #Result* query.

This mechanism is critical for compatibility. It allows any standard JDBC/ODBC client to execute complex recursive logic (and receive standard result sets) without requiring protocol changes, special driver support, or multiple round-trips.

Once the Dataflow script is invoked, the JIT Virtualization Runtime takes over to manage its execution.

4 THE JIT VIRTUALIZATION RUNTIME

The core innovation of the Dataflow architecture is the *Virtualization Runtime*. Unlike a static query plan, this runtime inspects the state of the data between statements to perform Just-In-Time (JIT) optimizations. Specifically, each statement within a loop body is executed individually, and the runtime rewrites queries and re-optimizes execution paths on *every single iteration* based on fresh statistics and virtualization states.

4.1 JIT Table Virtualization

A major hurdle in executing recursive workloads on distributed storage is the trade-off between *Latency* and *Throughput*.

Ocient's NVMe-based Compute-Adjacent Storage is optimized for massive throughput. However, for small datasets, the setup costs (specifically the distributed consensus, Raft [30], required to register table metadata, plus the overhead of the storage stack) dominate the execution time. For a graph traversal step involving only a few thousand rows, this overhead is prohibitive.

The runtime solves this via Virtualization. By trapping DDL and DML operations, it keeps small tables purely in the Coordinator's memory.

(1) **Interception:** When the script executes *CREATE TABLE #Queue*, the runtime registers a logical handle in memory without contacting the storage layer.
(2) **In-Memory Access:** Subsequent *INSERT* and *DELETE* operations are processed directly against this memory handle, bypassing the storage stack entirely. Reads against the table are handled by CTE injection (next item) rather than served from this handle.

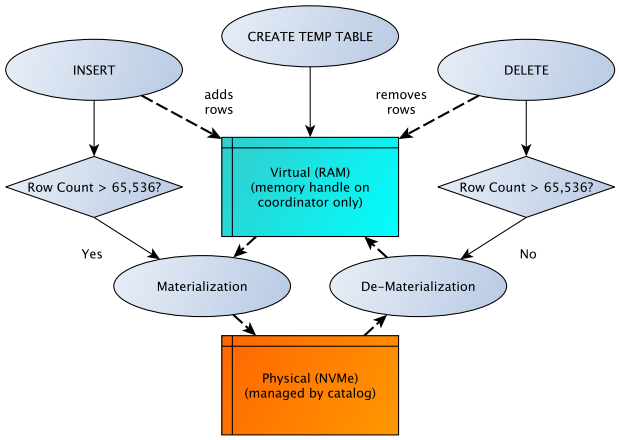

Figure 3: The JIT Virtualization Lifecycle: The runtime automatically transitions tables between RAM and NVMe based on row counts.

(3) **CTE Injection:** When a query references this table, the runtime injects the data as literals (using *UNNEST* arrays). At that point, it is treated as a standard result set that the optimizer may choose to broadcast, shuffle, or process locally depending on the plan.

Concretely, when a query references a virtualized table, the runtime rewrites it to inject the data as one constant array per column, fed through a single *UNNEST* that takes one element from each array per row (e.g., a join against *#Queue* becomes a join against *(SELECT UNNEST(col1, col2, ...))*). The Ocient optimizer treats the result like any other constant subquery and may broadcast, shuffle, or process locally. The *UNNEST* operation is lightweight, effectively turning the temporary table into a stream of literals without any storage I/O.

This technique utilizes the Ocient SQL engine's ability to optimize constant arrays. (In the Section 1.1 example, *#Q* virtualizes here while the BFS frontier is small.)

4.2 Thresholds and Materialization

The runtime enforces a memory safety limit. Our experiments demonstrated that while the parser can handle upwards of 250,000 items in a CTE, the optimal performance crossover point is lower.

- **Physical Crossover:** Pure I/O vs. parsing speed crosses over at approximately **128,000 rows**.
- **Throughput Stability:** In recursive loops, we observed significant degradation at 100,000 rows due to cumulative parsing overhead.

The default virtualization row limit is set to **65,536 rows** (64k), conservatively below both inflection points. If an *INSERT* exceeds this, the runtime flushes the data to NVMe, registers the table in the metadata Raft cluster, and switches to physical mode for the remainder of the query.

4.2.1 Sensitivity to Cluster Size and Hardware. The 65,536 figure is a default, not a hardcoded constant. Both the row limit and a complementary per-row byte budget are exposed as session-level mutable parameters (*dataflowVirtualizationMaxRows* and *dataflowVirtualizationMaxBytesPerRow*, default 10,000 bytes). The product (roughly 625 MiB by default) bounds the in-memory size of any single virtualized table on the Coordinator, and the runtime materializes whenever either bound is crossed. The row cap is never raised when rows are narrower than the byte default; it is only *lowered* when rows are wider, so the byte budget is a one-sided safety valve rather than a tuning knob to push virtualization further.

The threshold's value is largely independent of the compute cluster size and storage hardware. The Raft consensus that virtualization avoids is the *metadata* Raft, not a per-segment consensus on the Foundation Nodes; that quorum is small (typically a handful of voting members; on even-numbered SQL-Node deployments a tiny witness node brings the membership up to an odd quorum) and does not grow with the number of compute nodes. Adding Foundation Nodes increases throughput on materialized state but does not increase the per-DDL Raft round-trip cost that motivates virtualization in the first place. Per-row latency on NVMe is similarly bounded by SPDK ingest throughput, which is per-node rather than cluster-wide. As a result, the same threshold has worked across the two clusters this paper evaluates (Section 9); we have not separately swept it against additional network-bandwidth, NVMe-IOPS, or CPU-speed configurations.

We intend to conduct sensitivity analysis varying network bandwidth, NVMe IOPS, or CPU speed. The dominant cost the threshold is selected to avoid (metadata Raft round-trips) is latency-bound rather than bandwidth-bound, and NVMe latency is consistently low across commodity hardware generations. Dynamic auto-tuning of the threshold based on hardware telemetry is future work.

4.2.2 Memory Safety under Concurrency. Each Dataflow runs with its own virtualizer instance on the SQL Node it was issued against; virtualized state is process-local and is dropped when the Dataflow's *FINALLY* block runs (or upon cancellation). Per-table memory is bounded by the row-count and byte-budget caps above. There is no shared memory pool across concurrent Dataflows: total Coordinator-side memory used by virtualization is bounded by (#concurrent Dataflows) × (#virtual tables per Dataflow) × (byte budget per table), and concurrency is in turn bounded by the SQL Node's connection limits. The byte budget prevents unusually wide rows (e.g., large *VARCHAR* or array columns) from consuming far more memory than the row count alone would suggest.

4.3 Dynamic Dematerialization

A unique feature of this architecture is the ability to transition in both directions. In algorithms like K-Core Decomposition [6], the graph shrinks iteratively. A table that starts as massive (Physical) may eventually become small enough to fit in memory.

The runtime tracks *DELETE* operations. If a physical table's row count drops below the virtualization threshold (64k), **Dematerialization** is triggered:

(1) Execute *SELECT * FROM #PhysicalTable* to fetch the rows.
(2) Store the rows in the virtualizer.
(3) Drop the physical table asynchronously.

Thus, the tail of complex reduction algorithms runs with in-memory latency, even if the algorithm began with PBs of data. The Stable Marriage case study (Section 8) is a concrete example: rejected proposals are deleted from every iteration, and as the matching converges that table shrinks below the threshold and dematerializes.

4.4 Scalar & List Evaluation

For flow control, the engine pulls values out of the SQL engine into imperative variables. **Pure-imperative** updates (e.g., *SET @i = @i + 1*) are evaluated directly by the runtime. **Constant-folded** expressions resolved by validator-time propagation skip execution entirely; the validator returns the value. **Coordinator-local** plans cover the residual case: when the referenced state is virtualized in the Coordinator's memory, the optimizer plans the wrapped *SELECT* to run strictly on the SQL Node, skipping the dispatch to Foundation Nodes. Tight control loops therefore react to data state without distributed network overhead.

4.5 Adaptive Execution via Iterative Re-planning

A profound, albeit secondary, benefit of the Dataflow architecture is the introduction of **Adaptive Query Processing** (AQP) [8] to recursive workloads. Standard database engines typically plan a *WITH RECURSIVE* query as a single monolithic DAG. The optimizer must select join strategies (e.g., Broadcast vs. Shuffle) based on initial estimates of the recursive depth and fan-out. If these estimates prove incorrect (for example, a graph traversal encountering a supernode in iteration 3), the execution is locked into a suboptimal physical plan for the duration of the query. In the Section 1.1 example, this is the moment *#Q*'s join against *Edges* switches from broadcast to shuffle as the frontier grows, and back as it shrinks.

4.5.1 Just-In-Time Plan Evolution. Because the Dataflow runtime treats the loop body as a fresh SQL statement in each iteration, the Ocient optimizer re-evaluates the execution strategy dynamically based on the exact state of the traversal.

- **Exact Cardinality:** In Ocient, row counts for the *#Queue* table are exact and immediately available in the system metadata after the previous *INSERT* commits. The optimizer does not rely on stale histograms or estimates.
- **Strategy Switching:**
  - *Early Iterations (Low Selectivity):* When the frontier is small (injected via CTEs), the optimizer treats the data as constant arrays and typically chooses a *Broadcast Join*, replicating the queue to all Compute Nodes to avoid network shuffling of the massive edge table.
  - *Peak Iterations (High Selectivity):* As the traversal hits the dense part of the graph (materialized to storage), the optimizer automatically switches to a *Shuffle Hash Join*, optimizing for bandwidth over latency.

5 OPERATIONAL SAFETY & GOVERNANCE

Integrating recursion into a multi-tenant warehouse introduces significant risks regarding resource starvation and runaway execution. Ocient leverages the Dataflow architecture to enforce governance policies that are difficult to implement with monolithic operators.

5.1 Fairness via Release-Acquire Scheduling

A naive implementation of recursion (e.g., a kernel-level recursive union operator) typically monopolizes a WLM concurrency slot for the entire duration of the query. For deep graph traversals, this could block a slot for hours, potentially starving short-running queries in the same service class.

The Dataflow runtime avoids this by adopting a **Release/Acquire** pattern. Although the client connection remains open, the Dataflow itself does not hold a WLM concurrency slot during the "Control Phase". This phase consists of variable updates and loop-condition checks.

(1) **Acquire:** When the interpreter encounters a heavy statement (e.g., *INSERT INTO #Next...*), it submits a request to the WLM.
(2) **Execute:** The statement enters the queue (FIFO per service class). In high-concurrency environments, it competes fairly with other queries.
(3) **Release:** Upon statement completion, the WLM slot is immediately released before the interpreter proceeds to the next line of the script.

This ensures that a massive recursive job yields the cluster to other tenants between iterations, preventing head-of-line blocking. Each step of the recursion is treated as an independent job for scheduling purposes, ensuring that long-running recursive queries do not degrade the overall system responsiveness.

5.2 Intrinsic Spilling via Storage Tables

Traditional MPP systems often attempt to keep the recursive working set in memory to maximize performance, resorting to *spilling to disk* only as a slow exception path when RAM is exhausted. This often leads to Out-of-Memory (OOM) failures or severe performance degradation when the spill path is engaged.

In contrast, Ocient's *#Queue* and *#Result* tables are regular, distributed tables backed by the Compute-Adjacent Storage (NVMe). They are not memory-limited structures. As a side effect, no additional code complexity had to be added.

- **Scalability:** The system can handle intermediate frontiers of trillions of rows without code changes or special spill logic. The spill path is the normal path.
- **Performance:** Ocient is optimized for NVMe throughput, utilizing SPDK for kernel-bypass I/O. Thus, the latency penalty of writing intermediate states to disk is minimal compared to the safety guarantees this path provides.

5.3 Runaway Recursion Prevention

Imperative loops introduce the risk of infinite cycles. To protect the cluster, we enforce a hierarchical safety strategy that places power in the hands of both the administrator and the user.

System-Level Hard Limit: A global configuration parameter acts as a circuit breaker for loops. If the count is exceeded, the runtime throws an exception to prevent infinite resource consumption.

User-Defined Constraints: Users can override system limits to impose stricter limits on themselves using specific grammar constructs, such as appending `MAX ITERATIONS` directly in the `WHILE` or `FOR` loop syntax. This allows developers to write safe recursive logic without relying on external timeouts.

Cancellation Propagation: Finally, if a user cancels the parent Dataflow request, the runtime identifies the currently executing child statement and issues a kill signal. The resulting exception bubbles up to the Dataflow interpreter, which triggers the *FINALLY* block. This guarantees that temporary tables are dropped and storage is reclaimed, even during an abrupt user cancellation.

6 STORAGE, INDEXING, AND RESILIENCY

A distinct characteristic of the Ocient architecture is its rejection of user-defined data partitioning for workloads. While many specialized graph systems advocate for complex partitioning schemes (e.g., METIS [21] or Label Propagation [34]) to co-locate connected subgraphs on the same physical node, we argue that such strategies are operationally complex at hyperscale.

6.1 Storage and Resiliency: Discussion

The case against static co-location. At petabyte scale, hardware failures are not anomalies. Rigid co-location (e.g., hashing edges by source id to keep neighbors local) creates operational hazards:

(1) **Resiliency vs. Locality:** If a node fails, its data must be served from another node, immediately breaking the co-location invariant.
(2) **The Hotspot Problem:** Real-world graphs (social networks, web graphs) follow power-law distributions [5]. Static partitioning produces severe skew where high-degree vertices overwhelm specific worker nodes.

Indexing strategy. While we eschew partitioning, we heavily leverage indexing to minimize NVMe scan overhead during the expansion phase of recursion.

- **Secondary indexes:** For highly selective traversals (e.g., k-hop reachability from a single node), the transpiler leverages secondary indexes on the Edge table's *SourceID* and *DestID* columns. This allows the Foundation Nodes to perform index-lookups rather than full table scans.
- **Zone Maps:** For analytical graph algorithms (e.g., PageRank or Weakly Connected Components [31]) that touch a significant fraction of the graph, the system bypasses indexes and utilizes Zone Maps (Min-Max pruning) to scan the Edge table at line-rate. This approach mirrors the design of modern column-store optimizations described in systems like XPRS [37] but adapted for modern NVMe substrates.

Mitigating skew via iterative resets. In static graph partitioning strategies, data skew is often compounding. If a high-degree vertex with millions of edges is hashed to Node A, Node A becomes a straggler. In recursive workloads, if the children of that vertex are also hashed to Node A (to preserve locality), the skew intensifies with every iteration, eventually crashing the node [11].

Ocient avoids the compounding effect by sidestepping locality-based partitioning altogether: per Section 6.1 we do not co-locate edges or vertices by hash. The Dataflow architecture then exposes a useful side-effect of the storage layer's default placement policy. Because the runtime treats each iteration's recursive member as a fresh *CREATE OR REPLACE TABLE AS SELECT*, the transition from

#Queue (Iteration N) to *#Next* (Iteration $N + 1$) flows through the standard storage write path:

(1) **Execution:** During the join, a specific worker may process a disproportionate number of edges (the supernode).
(2) **Materialization:** The output rows are written to the Distributed Storage Layer (NVMe) using the storage layer's default placement policy, which assigns blocks of rows to nodes in a round-robin fashion regardless of value-level locality.

The next iteration therefore consumes input that is approximately uniformly distributed across the cluster's storage, even if the prior iteration suffered from straggler workers. We emphasize that round-robin placement is a pre-existing property of Ocient's append-only writer rather than a Dataflow-specific optimization; the contribution here is recognizing that the per-iteration *CREATE TABLE ... AS SELECT .../RENAME* pattern naturally inherits it. To verify this empirically, we instrumented a separate 100-billion-edge WCC dataflow run with per-iteration timing exposed via *sys.completed_queries* polling between child statements (slower than the run in Table 3 because of the polling and an extra per-iteration count guard; the value here is the iteration-stability signal, not the total time). Across all 22 iterations of the converging algorithm, per-iteration wall time was stable at 290 ± 5 s, with the dominant join CTAS holding at ∼150 s every iteration; persistent supernode skew would have inflated iteration time as the worst-loaded node fell further behind, and we observe no such growth. The same instrumentation on the trillion-edge graph and a per-foundation-node breakdown of the join work are left for future work.

Scheduler-driven operator spilling. Unlike memory-first graph frameworks which often suffer from cliff-edge performance where running out of RAM causes job failure, Ocient treats NVMe as a first-class extension of the memory hierarchy. The system employs a sophisticated, scheduler-driven spilling mechanism to handle memory pressure during massive graph joins.

When a parallel instance of an operator (e.g., a Join builder) fails to allocate memory from the global pool, it signals an OOM condition to the central scheduler. The scheduler evaluates the global state and selects specific operator instances to spill, often from the lowest priority query.

If a Hash Join instance is selected to spill, it transitions from a streaming execution mode to a **partitioned execution mode**.

(1) **Total Flush:** The operator writes its entire current in-memory hash table to local NVMe, reorganizing the data into buckets based on hash keys.
(2) **Mode Switch:** The operator stops building an in-memory structure. Instead, subsequent incoming rows (from both build and probe sides) are buffered into these partitions.
(3) **Recursive Spilling:** If the buffers for these partitions consume too much memory, the scheduler may signal the operator to flush specific buffers to disk again.

Once the input is fully consumed, the operator processes one bucket at a time, loading it back into memory. This allows recursive queries on petabyte datasets to succeed even when the working set of a single join iteration vastly exceeds the aggregate RAM of the cluster, effectively trading execution time for reliability.

7 ARCHITECTURAL POSITIONING & FUTURE STANDARDS

The database market is currently bifurcated between specialized Graph Databases (LPGs) and general-purpose Distributed Compute Frameworks. Ocient's Dataflow architecture occupies a unique middle ground.

7.1 Ocient vs. Native Graph Databases (LPGs)

Systems like Neo4j or TigerGraph [9, 12] are optimized for *index-free adjacency*, utilizing physical pointers to traverse edges.

- **The Trade-off:** Pointer traversal is well-suited for deep, narrow queries (e.g., "Find the shortest path between A and B"). However, it suffers from poor memory locality and random-access patterns when processing analytical queries (e.g., "Calculate centrality for 1 billion nodes").
- **Ocient's Niche:** Ocient targets Analytical Graph (OLAP) workloads. By using set-based Relational Algebra (Joins) rather than pointers, we sacrifice latency on single-path lookups to gain massive throughput on whole-graph analytics.

7.2 Ocient vs. Disaggregated Compute Frameworks

Apache Spark GraphX [40] is a standard for distributed graph analytics. While it effectively utilizes memory for performance, its architecture faces a penalty when working sets exceed RAM.

- **The Serialization Wall:** When memory-based frameworks like Spark spill to local disk, they incur a heavy serialization tax: converting JVM objects into serialized byte streams and passing them through the OS file system. This context switching and serialization overhead often dominates the execution time for massive graphs.
- **Zero-Copy Paging:** Ocient's architecture leverages NVMe not just as a spillover, but as the primary storage medium. Data on NVMe is stored in the engine's native binary format. Spilling intermediate results leverages zero-copy Direct Memory Access (DMA), handled by SPDK to bypass the OS kernel and eliminate serialization overhead. While complex internal structures (e.g., hash maps) require conversion to this format, the system minimizes serialization overhead and allows Ocient to sustain performance on graphs larger than RAM without the performance cliff typical of managed-memory frameworks.

7.3 Architecture for SQL:2023 and GQL

With the Dataflow runtime established as a procedural superset of SQL, Ocient is well-positioned to support emerging standards. Both the ISO/IEC GQL standard [19] and the Property Graph Queries (SQL/PGQ) extension to SQL:2023 [18] introduce complex pathing constraints that map cleanly to our imperative execution layer. In our future work, we intend to implement fixed-length and quantified pattern matching, shortest-path modes, *GRAPH_TABLE* projection, graph DML, and composable subquery expressions.

- **Quantified Paths:** The full GQL/SQL/PGQ quantification surface is in scope, including bounded (*{m,n}*), unbounded

(*, +), and exact-count quantifiers, applied to either edge or path patterns. Each form transpiles to a *WHILE* loop whose iteration count is governed by the quantifier; node and edge constraints become per-iteration filters on a queue-typed temp table. The illustrative pattern *(a)-[:KNOWS]->{1,5}(b)* is one instance of this mapping rather than the only supported shape.
- **Path Restrictions:** Standards-defined modes like *TRAIL* (no repeated edges) or *ACYCLIC* (no repeated nodes) can be implemented by adding a "history" array column to the Dataflow's intermediate queue, leveraging Ocient's native support for large array types. *ANY SHORTEST*, *ALL SHORTEST*, and *ANY* path-selector modes map to short-circuit checks already idiomatic in our case studies (cf. the bidirectional-search example).
- **Composable Subqueries:** The GQL *EXISTS*, *COUNT*, and *COLLECT* subquery expressions, which themselves may contain pattern matching, transpile to nested Dataflow blocks invoked from the outer query's expression evaluator.

By targeting the imperative Dataflow rather than the physical plan directly, Ocient can adopt these new parsers as they mature without requiring fundamental changes to the execution kernel.

7.4 Why a User-Visible Imperative Layer?

An alternative is to keep the user-facing language declarative (an extended recursive CTE or operator) and push adaptive decisions (state representation switching, per-iteration replanning, threshold-based virtualization) into the executor as in adaptive recursive query optimization [8]. We considered this and rejected it for our initial implementation due to three concrete reasons drawn directly from our case studies in Section 8:

(i) Expressivity, not just optimization. Some of the workloads we serve are not expressible in standard *WITH RECURSIVE* with good performance. The SQL recursion semantics require monotonicity (no *DELETE* of intermediate state, ruling out Stable Marriage), linearity (the recursive name appears at most once in the recursive member's *FROM*, ruling out the two-frontier coupling required for Bidirectional Search early-termination), and have no standard mechanism for floating-point convergence checks (PageRank's ϵ-stop). An adaptive optimizer over a declarative CTE can re-plan but cannot relax language-level constraints; the imperative surface gives users expressive power the declarative form cannot match.

(ii) Inspectability between iterations. Several Dataflow optimizations rely on observing the actual size and content of intermediate state *between* iterations and routing the next iteration accordingly: virtualizing when the frontier is small, dematerializing when it shrinks, adapting join strategies when cardinality flips. Implementing these inside a declarative recursive operator requires either (a) special-casing that materializes after every iteration anyway (losing the encapsulation that motivated the declarative form), or (b) lifting per-iteration boundaries into the optimizer, at which point the operator is a procedural script in disguise. Making that script first-class lets the user inspect, modify, and reuse it.

(iii) Operational integration. The release/acquire WLM pattern (cf. Section 5.1), the *FINALLY*-block cleanup, and the deterministic per-iteration boundaries that admit cancellation, runaway-loop limits, and the "Hot Swap" protocol (cf. Section 3.3) all rely on the loop body being a sequence of independent, schedulable statements. A monolithic recursive operator forfeits these.

The procedural surface gives users access to algorithms they could not otherwise express. The JIT executor makes those algorithms cheap to run.

8 ALGORITHMIC CASE STUDIES

While the transpiler handles standard SQL recursion, the true power of the Dataflow architecture is revealed when users write Dataflows manually to implement logic that is prohibitively restricted by the SQL standard (e.g., lack of support for non-monotonic operations).

8.1 Bidirectional Search

Finding the shortest path between two specific nodes (A and Z) in a massive graph is inefficient with a standard BFS starting from A. The frontier expands exponentially in $O(b^d)$, where b is the branching factor and d is the depth.

This is particularly acute in scale-free networks (graphs following a power-law distribution like social networks), where hitting a high-degree hub node causes the search frontier to explode in size. A much more efficient approach is **Bidirectional Search**: expanding from both A (forward) and Z (backward) simultaneously until the frontiers meet. The complexity drops to $O(b^{d/2})$, which can mean the difference between seconds and hours.

Standard SQL *WITH RECURSIVE* cannot express this algorithm efficiently due to constraints on linear recursion. It comes either with a cycle of dependencies or with a self-join of the recursive set, which is typically not supported by query optimizers. Even if supported, the query cannot terminate *early* (i.e., stop immediately when the frontiers touch) because the WHERE clause of one recursive branch cannot depend on the dynamic state of the other branch.

In an Ocient Dataflow, we can maintain two distinct temporary tables (*#FrontierFwd*, *#FrontierBwd*) that are seeded from the source and sink nodes. The algorithm alternates expand-and-test steps inside the control loop: expand forward, intersect with the backward frontier, update a found flag; if still not found, expand backward, intersect again. The intersection count lives in the control plane, so the loop ends the moment the frontiers touch and a path is found.

8.2 Iterative Matching (Stable Marriage)

Complex graph algorithms often require mixing constructive operations (adding to a set) with destructive operations (pruning a set). A classic example is the **Stable Marriage** problem, used in markets to match supply with demand. This requires an iterative cycle of proposing (Insert) and rejecting (Delete).

Standard SQL recursion is typically monotonic (additive only). Implementing rejection logic usually requires complex window functions to filter out invalid previous states. In a Dataflow, this can be expressed naturally as an imperative loop.

The full implementation is in the artifact repository. The body alternates four phases inside a loop: (1) free men propose to their highest-ranked remaining preference; (2) women evaluate suitors and reject all but the lowest-id man; (3) the engine counts the rejections into a control-plane scalar; (4) if non-zero, rejected pairs

are deleted from the proposal and preference tables and the men return to the free pool. The loop exits when no rejections remain. The *DELETE*-inside-loop pattern is what makes this non-monotonic and outside the standard *WITH RECURSIVE* surface.

8.2.1 The Virtualization Trade-off for Deletes. This example highlights a specific performance characteristic of the virtualization runtime. Because Ocient optimizes for append-only storage, tables do not enforce primary keys. Consequently, a virtualized *DELETE* operation must often be executed via an *EXCEPT ALL* set difference to identify surviving rows based on full-column comparison. This is computationally more expensive than a pointer-based delete in a native graph database.

However, the architectural advantage lies in the *lifecycle* of the table. By keeping tables like *#Rejections* virtualized, the system avoids any latency in the storage layer and the associated metadata (including cluster-wide synchronization) for the *CREATE*, *TRUNCATE*, and *DROP* operations that occur in every iteration of the loop. The slightly slower *DELETE* is accepted in exchange for near-instantaneous metadata operations and fast *INSERT* performance, yielding a lower total runtime for the algorithm.

8.3 Weakly Connected Components (Connectivity)

Beyond search and optimization, Dataflows support iterative analytical algorithms like **Weakly Connected Components (WCC)** via Label Propagation [31, 34]. This algorithm requires every node in the graph to repeatedly exchange IDs with its neighbors to determine subgraph membership.

We omit the full Dataflow body for space; the full implementation is in the artifact repository. The body initializes a *#Labels* table where every node's label is its own ID, then loops: each iteration joins the edges against *#Labels* to compute, for every source vertex, the minimum label of any neighbor (written into *#Updates*); a *CREATE OR REPLACE* on *#Labels* applies the smaller of the current and proposed labels; a scalar count of strictly improved labels is bound to *@changes* as the loop guard. Iteration ends when no label changes (*@changes = 0*). The cost shape is one large join plus one group-by per iteration; the runtime's JIT virtualization keeps both *#Updates* and *#Labels* in their best representation as the frontier evolves.

This pattern demonstrates the system's stability. Unlike a path-finding query that touches a small subset of data, WCC touches the *entire* dataset in every iteration. The Dataflow runtime can seamlessly swap *#Labels* to NVMe, ensuring the algorithm completes even when label state exceeds aggregate cluster memory, a common failure mode in memory-only graph frameworks.

8.4 PageRank (Centrality)

A key advantage of Dataflows over Recursive SQL is the ability to handle **floating-point convergence checks**. Standard recursion typically relies on a fixed iteration depth or an empty set return. Algorithms like PageRank, however, require iterating until the delta between iterations drops below a mathematical epsilon (e.g., 10^{-6}).

Algorithms like PageRank [32] usually require updating the entire state vector in every iteration. Ocient's Dataflow runtime optimizes this via a **Swap Pattern**: the new rank vector is materialized as a fresh NVMe-backed table, and the old table is dropped.

Again, we omit the full Dataflow body; the full code is in the artifact repository. The body declares a *@damping* factor (0.85), an *@epsilon* convergence threshold (10^{-6}), and a running *@diff* scalar; it initializes *#PageRank* to the uniform distribution and then loops while *@diff > @epsilon*. Each iteration computes the next rank vector as a single streaming *CREATE OR REPLACE TABLE #NewRank AS SELECT … GROUP BY e.dest* (the canonical PageRank join against *Edges*), updates *@diff* as the L1 norm of the rank delta via a scalar subquery, and swaps *#PageRank* to point at the new table. The control-plane scalar arithmetic on *@diff* alongside the data-plane GROUP BY is exactly the floating-point convergence pattern that standard *WITH RECURSIVE* cannot express.

The ability to perform scalar math on the *@diff* variable on the Coordinator while performing the massive aggregation on Foundation Nodes enables Ocient to execute iterative numerical methods efficiently.

9 EVALUATION

9.1 Experimental Setup

Experiments 1 through 4 were conducted on an Ocient cluster consisting of **2 SQL Nodes** (Coordinators), **10 Foundation Nodes** (Compute), and **1 Loader Node**.

- **Processors:** Dual-socket Intel Xeon Gold 6140 (Coordinators) and Gold 6148 (Compute/Loader).
- **Memory:** 768 GB RAM per node.
- **Node Roles:** Foundation nodes handle query execution, while Loader nodes are responsible for data being written to disk, including from *INSERT* and *CTAS* statements.

This setup ensures that all physical DDL operations (Create/Drop Table) must pass through distributed consensus (Raft), providing a realistic measurement of metadata overhead in a production environment. (Note: Experiment 5 utilizes a distinct hyperscale hardware configuration, detailed in Section 9.6). Experiments 1–4 each function as a targeted ablation of one core Dataflow mechanism, as described in the Analysis paragraph of each experiment. The contribution of **release/acquire WLM** scheduling and **"Hot Swap" overhead** to total runtime has not yet been isolated through controlled experiments and is a focus of future work.

9.2 Experiment 1: Latency & JIT Virtualization

We compared implementations of a linear chain traversal at shallow depths, i.e. 10–100 iterations (cf. Figure 4).

(1) **JIT Dataflow:** Uses virtualization and CTE injection.
(2) **Client Baseline:** A Java client issuing *CREATE TABLE AS SELECT* for each step.
(3) **Unrolled SQL:** A single, monolithic *SELECT* statement with *N* explicit joins.

Analysis: The Client Baseline demonstrates the prohibitive cost of physical DDL in a Raft environment (~4.6 s per step). It involves the execution of a real *CREATE TABLE AS SELECT* per iteration

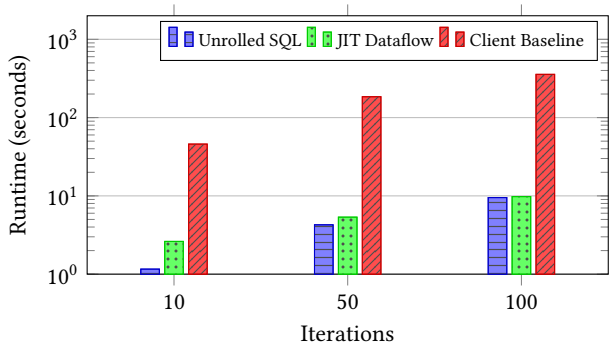

Figure 4: Shallow Iteration Latency

with no virtualization, no "Hot Swap", and no Dataflow-level re-planning. The **JIT Dataflow** reduces this by 97% via virtualization. While slightly slower than the unrolled version at shallow depths due to parsing overhead, the performance penalty is empirically capped at roughly 2.3× for our tested range (2.62 s vs. 1.16 s at 10 iterations). This relative overhead rapidly diminishes as the iteration count increases, closing to near parity (1.03×) by 100 iterations. The Dataflow runtime maintains latencies well below the physical alternative while dynamically handling states without the need for manual unrolling.

9.3 Experiment 2: Deep Recursion Scalability

We extended the comparison between **JIT Dataflow** and **Unrolled SQL** to 600 iterations to test the limits of the query optimizer.

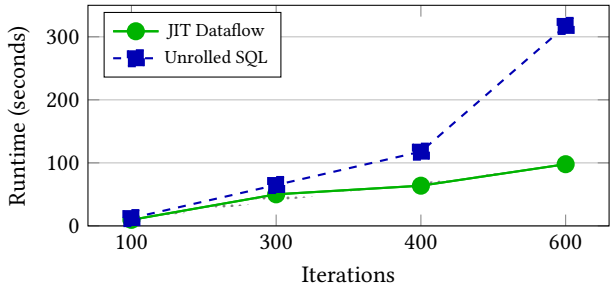

Figure 5: Deep Recursion Scalability

Analysis: Unrolled SQL exposes the same algorithm to the optimizer as one large monolithic DAG; the planner degrades super-linearly as it struggles with the complexity of hundreds of joins. The Dataflow maintains linear scalability by treating each iteration as an independently re-planned statement with a small number of joins. At 600 iterations, the Dataflow is **3.2× faster** (cf. Figure 5).

9.4 Experiment 3: Throughput Scaling & Thresholds

We ran a fixed 5-iteration recursive loop while scaling the payload from 1,000 to 10 million rows to quantify the behavior of the Virtualization Runtime as it approaches and exceeds the **65,536 row** threshold (cf. Table 1).

Table 1: Throughput Scaling (5-Iteration Loop)

Rows	Total Time	Time Per Row	Mode
1,000	8.70 s	1,741,542 ns	Virtual
10,000	17.50 s	350,022 ns	Virtual
100,000	**49.46 s**	**98,929 ns**	**Transition**
1,000,000	42.50 s	8,503 ns	Physical
10,000,000	48.63 s	972 ns	Physical

Analysis: The data reveals a performance inflection point at 100,000 rows. Here, the row count exceeds the virtualization limit (65,536), forcing the runtime to materialize tables to NVMe. However, the data volume is too small to efficiently amortize the fixed costs of Raft consensus and disk I/O, resulting in the worst-case total time (49.46 s). As the volume increases further to 1M and 10M rows, the system enters its optimal throughput zone. The time per row drops drastically (from 99k ns to 972 ns), and the total execution time remains nearly flat despite a 100× increase in data. This isolates the virtualization vs. Raft-DDL trade-off across several orders of magnitude. It also confirms that Ocient's physical path is very efficient at scale, even if it incurs a latency penalty for small, super-threshold datasets.

9.5 Experiment 4: Dynamic Dematerialization

To validate the system's ability to recover performance as data shrinks (e.g., during reduction algorithms), we executed a reduction dataflow that begins with 200,000 rows (Physical) and iteratively deletes 50% of the data until the table is empty. We measured the server-side duration of each step.

Table 2: Dematerialization Trace (Reduction Dataflow)

Row Count	Step Duration	State
200,000	15,983 ms	Physical (NVMe)
99,999	20,214 ms	Physical (NVMe)
49,998	**10,109 ms**	**← Dematerialization**
24,998	9,971 ms	Virtual (RAM)
12,498	4,129 ms	Virtual (RAM)
3,123	2,637 ms	Virtual (RAM)

Analysis: A clear performance improvement is visible when the row count drops below the 65,536 threshold. The step time drops from ~20 s (physical) to ~10 s (virtual), and then settles at ~3 s for small data. This validates the efficacy of **Dynamic Dematerialization**: as complex reduction algorithms proceed, the system automatically transitions from "Batch Throughput" to "Low Latency" mode, avoiding the penalty of disk I/O for the long tail of the computation. Had the workload remained in Physical mode after dropping below threshold, runtime would extrapolate from the 99,999-row step (~20 s). The observed 4 s at the 12,498-row virtual step (Table 2) isolates the dematerialization mechanism's contribution.

9.6 Experiment 5: Hyperscale WCC Performance

To evaluate the system's performance on hyperscale graphs, we generated synthetic graphs with a Zipfian distribution (Power Law) and ran the Weakly Connected Components algorithm. To accommodate the massive scale, this experiment utilized a distinct, higher-capacity cluster configuration equipped with dual-socket AMD EPYC 9654 96-Core Processors and 2.3 TB of RAM (24 x 96 GB) per node. We compared three configurations:

(1) **Spark GraphFrames:** A 5-node cluster with Apache Spark.
(2) **Ocient Legacy:** Ocient using external SQL driver orchestration (JDBC).
(3) **Ocient Dataflow:** The Dataflow-optimized runtime on an 11-node cluster (8 Compute, 2 Loader, 1 SQL).

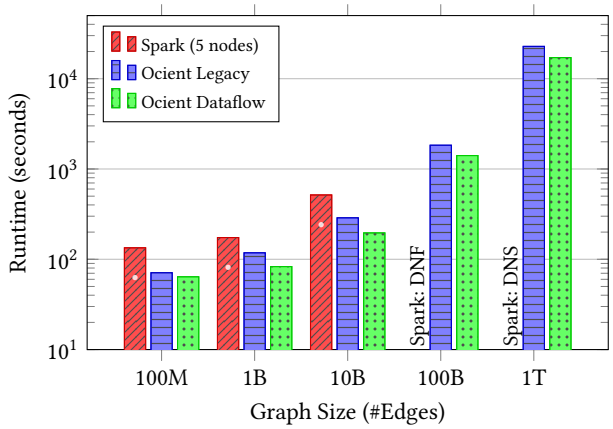

Figure 6: WCC Runtime Scalability (log-scale comparison across graph sizes). DNF indicates the framework Did Not Finish, while DNS means we Did Not Start it.

Analysis: While the Ocient cluster possessed approximately 2× the hardware resources of the Spark cluster (11 nodes vs. 5), the performance differential grows with scale and far exceeds this hardware ratio at the largest sizes. At **100 billion edges**, the memory-only framework (Spark) failed to complete (Did Not Finish) due to executor memory pressure and shuffle-spill local-disk exhaustion, while Ocient completed the task in roughly 23 minutes.

9.6.1 Normalized Throughput per Node. To address the cluster-size disparity, Table 3 reports per-node throughput (edges processed per node per second). The Spark numbers in this table come from runs under the same $\alpha = 1.5$ Zipfian distribution as the Ocient cluster measurements, so the rows are directly comparable. At 100M edges (where the working set fits comfortably in RAM on either system), the per-node throughputs of Spark and Ocient Dataflow are within a few percent of each other (0.149M vs. 0.142M edges/node-s). At 1B edges Spark slightly leads Ocient on a per-node basis (1.15M vs. 1.10M); this is the regime where Spark is most competitive and where the architectural advantages of Ocient have least to offer. The Ocient advantage emerges as the working set grows: at 10B edges Ocient delivers 1.2× the per-node throughput of Spark, and at 100B and 1T edges Spark cannot complete at all, making per-node comparison undefined. The interpretation is that the "2× hardware" framing under-states the difference at scale: Ocient's NVMe-first design extends the runnable envelope, not just shortens runtime within it.

Table 3: WCC per-node throughput (M edges / node / s) under the unified $\alpha = 1.5$ Zipfian distribution.

Edges	Spark (5n)	Legacy (11n)	Dataflow (11n)
100M	0.149	0.128	0.142
1B	1.15	0.770	1.10
10B	3.88	3.16	4.64
100B	DNF	4.96	6.46
1T	DNS	3.99	5.31

We do not present a per-dollar comparison: list prices for Ocient and a comparable Spark deployment depend heavily on commitment terms and storage tier choices, and any single point comparison would mislead. The per-node table is the most defensible architectural signal we can extract from these runs.

9.6.2 Comparison vs. Single-Node Recursive Engines. A natural question is how the Dataflow runtime positions against systems with strong single-node recursive query support, specifically Umbra [29] and DuckDB [33]. We did not have lab hardware for either, so we ran them on GCP using `n2d-highmem-96` VMs (96 vCPU AMD EPYC Milan, 755 GB RAM), the largest AMD-EPYC shape we could obtain without a quota-increase ticket. This is still smaller than one Ocient EPYC 9654 node (192 physical cores, 2.3 TB RAM) but is the same vendor and family (Zen 3 Milan vs. Zen 4 Genoa). All measurements use the same label-propagation WCC algorithm and the same Zipfian input ($\alpha = 1.5$) as the cluster runs.

On this VM, Umbra is consistently faster than DuckDB where both fit in memory, but Umbra has no spill path and OOMs at 10B while DuckDB completes via its spill-to-disk implementation. The single-node WCC harnesses for both engines are bundled in the artifact repository under *data/*.

Extrapolation methodology. To project the GCP measurements onto a single Ocient EPYC 9654 node, we apply a per-node hardware correction of 5×. This is the product of three factors set at the most competitor-favorable end of each defensible range, so the extrapolated competitor runtime is a lower bound rather than an expected value: (i) 4.0× physical-core ratio (192 vs. 48 cores); (ii) 1.15× IPC uplift, the top of the published Zen 3 → Zen 4 range on integer/SQL workloads; (iii) 1.09× per-core memory-bandwidth uplift, derived from 12-channel DDR5-4800 (Genoa) vs. 8-channel DDR4-3200 (Milan) normalized by the respective core counts. The product is $4.0 \times 1.15 \times 1.09 \approx 5.0\times$. The same correction applies to DuckDB and Umbra. Neither has a native MPP execution path, so we do not extrapolate either to a cluster: even with this generous per-node correction, the Ocient comparison column is the engine running on 11 such nodes, an 11:1 hardware advantage that single-node engines fundamentally cannot match.

What the comparison shows. Table 4 extrapolates the GCP measurements onto a single Ocient EPYC 9654 node. At 100M edges

Table 4: DuckDB and Umbra WCC runtime extrapolated onto a single Ocient EPYC 9654 node (192 physical cores, 2.3 TB RAM). Extrapolation: `n2d-highmem-96` measurement divided by 5×. Ocient figures from Table 3 are reproduced for reference; neither single-node engine has an MPP execution path so a cluster-equivalent column is not defined.

Edges	DuckDB 1n *(extrap.)*	Umbra 1n *(extrap.)*	Ocient 11n *(meas.)*
100M	2.6 s	1.7 s	64 s
1B	25 s	19 s	83 s
10B	374 s	OOM	196 s
100B	OOM	OOM	1,408 s
1T	OOM	OOM	17,126 s

both single-node engines crush 11-node Ocient on equivalent-spec hardware (DuckDB 25×, Umbra 38×), the regime where in-process compiled engines dominate because the working set fits in cache plus RAM. At 1B edges per-node Ocient is still slower than either single-node engine even at the maximum plausible per-node correction. At 10B edges DuckDB on one EPYC 9654 node is roughly 1.9× slower than the 11-node Ocient cluster (Umbra OOMs entirely): the spill-path tax has become throughput-limiting and even the most favorable extrapolation cannot recover. At 100B and 1T edges all single-node engines exceed available RAM; Ocient is the only configuration in the table that runs those scales at all.

10 RELATED WORK

The challenge of integrating recursion into data management systems has a rich history.

Language Standards: The SQL:1999 standard introduced *WITH RECURSIVE* [17]. Optimization techniques for linear recursion such as Magic Sets [4], semi-naive evaluation [3], and counting-based termination have been studied extensively, but typically assume a single-node context. The ISO/IEC GQL [19] and SQL:2023 SQL/PGQ [18] standards represent the industry's move toward declarative graph patterns. While standards define the declarative interface, vendor extensions have explored imperative execution. For example, complex imperative execution via compound statements has long been present in enterprise systems like IBM DB2 [16], though typically not utilized as an ephemeral execution substrate for recursive (graph) analytics.

MPP and Adaptive CTE Optimization: Distributed engines have repeatedly grappled with how to schedule recursive CTEs alongside set-oriented MPP operators. Vertica [22], Greenplum [25], and Snowflake [7] support *WITH RECURSIVE* but document caveats around plan stability and state spillage at scale; broader work on optimizing CTE evaluation in shared-nothing settings is summarized in surveys of MPP optimization [35]. Architecturally, the closest comparable feature is Vertica's iterative use of internal temp tables only. We benchmarked Vertica Community Edition, but its EULA prohibits sharing the measured results without prior written consent; we therefore position against Vertica architecturally only. Adaptive Query Processing [2, 8] re-plans *within* a single declarative query in response to runtime cardinalities, and Ocient's per-iteration replanning (cf. Section 4.5) is closely related: every iteration of a Dataflow loop is, in effect, an independent declarative SQL statement that is re-optimized against fresh exact cardinalities. Where AQP folds adaptivity into a monolithic operator, our approach surfaces the iteration boundary as a first-class language construct in the explicit imperative loop, which enables more expressivity, inter-iteration inspectability, and operational integration. We see the two as complementary: an adaptive declarative executor is a strong default for monotonic linear recursion while Dataflows cover additional workloads that fall outside it.

Distributed Graph Frameworks: BSP-style systems (Pregel [26], Spark GraphX [13, 40]) and timely/differential dataflow [27, 28] target a different deployment model than ours: in-memory single-tenant execution rather than HA MPP, and they incur RDD or fixed-point abstraction overhead for the fine-grained per-iteration work that the Dataflow surface exposes directly. Streaming systems like Apache Beam [1] are unrelated: they target unbounded out-of-order processing rather than bounded relational recursion.

11 CONCLUSIONS AND OUTLOOK

By virtualizing small recursion state as inline CTEs and transitioning to NVMe-backed tables as frontiers grow, the Dataflow runtime supports standard *WITH RECURSIVE* and iterative graph analytics at trillion-edge scale within a single MPP execution surface.

Several avenues remain for future work in optimizing the performance characteristics of the Dataflow runtime. We intend to push scalability limits further via detailed instrumentation on trillion-edge graphs and per-node breakdowns to provide deeper insights into processing overhead in our implementation. We plan to develop robust dynamic auto-tuning for the virtualization threshold based on real-time hardware telemetry—while avoiding oscillating behavior for materialization/dematerialization. For standards compliance, work is ongoing to implement comprehensive SQL:2023 and GQL support including pattern matching, shortest-path modes, *GRAPH_TABLE* projection, and graph DML.

TRADEMARKS

Ocient, Ocient Hyperscale Data Warehouse, Compute Adjacent Storage Architecture, OcientGeo, and OcientML are trademarks of Ocient, Inc. in the United States and other countries. Other company, product, and service names might be trademarks, or service marks, of Ocient or other companies. All trademarks are copyright of their respective owners.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[2] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 261–272. https://doi.org/10.1145/342009.335420

[3] François Bancilhon. 1986. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*. Springer-Verlag, New York, NY, USA, 165–178.

[4] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,*

March 24-26, 1986, Cambridge, Massachusetts, USA. ACM, 1–15. https://doi.org/10.1145/6012.15399

[5] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (Oct. 1999), 509–512. https://doi.org/10.1126/science.286.5439.509

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). Retrieved June 29, 2026 from http://arxiv.org/abs/cs/0310049

[7] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[8] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[9] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). Retrieved June 29, 2026 from http://arxiv.org/abs/1901.08248

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

[11] David J. DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM* 35, 6 (1992), 85–98. https://doi.org/10.1145/129888.129894

[12] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 1433–1445. Retrieved June 29, 2026 from https://doi.org/10.1145/3183713.3190657

[13] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 599–613. Retrieved June 29, 2026 from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[14] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[15] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[16] IBM 2026. *DB2 13 for z/OS: SQL Reference - Compound Statement*. IBM. Retrieved June 29, 2026 from https://www.ibm.com/docs/en/db2-for-zos/13.0.0?topic=pl-compound-statement

[17] ISO/IEC 1999. *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. ISO/IEC. ISO/IEC 9075-2:1999.

[18] ISO/IEC 2023. *Information technology — Database languages — SQL — Part 16: Property Graph Queries (SQL/PGQ)*. ISO/IEC. ISO/IEC 9075-16:2023.

[19] ISO/IEC 2024. *Information technology — Database languages — GQL*. ISO/IEC. ISO/IEC 39075:2024.

[20] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. VERTEXICA: Your Relational Friend for Graph Analytics! *Proc. VLDB Endow.* 7, 13 (2014), 1669–1672. https://doi.org/10.14778/2733004.2733057

[21] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[22] 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. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801.

[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[24] Leonid Libkin. 2003. Expressive power of SQL. *Theoretical Computer Science* 296, 3 (2003), 379–404.

[25] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2530–2542. https://doi.org/10.1145/3448016.3457562

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[27] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. Retrieved June 29, 2026 from http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[28] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[29] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. Retrieved June 29, 2026 from https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html

[30] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 305–319. Retrieved June 29, 2026 from https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[31] Jean Francois Pacault. 1974. Computing the Weak Components of a Directed Graph. *SIAM J. Comput.* 3, 1 (1974), 56–61. https://doi.org/10.1137/0203005

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. Retrieved June 29, 2026 from https://www.semanticscholar.org/paper/The-PageRank-Citation-Ranking-%3A-Bringing-Order-to-Page-Brin/eb82d3035849cd23578096462ba419b53198a556

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[34] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E* 76, 3 (Sept. 2007). https://doi.org/10.1103/PhysRevE.76.036106

[35] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 337–348. https://doi.org/10.1145/2588555.2595637

[36] 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), 21. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 03.-07, März 2025, Bamberg, Germany, Proceedings (LNI)*. Gesellschaft für Informatik e.V., 679–690. https://doi.org/10.18420/BTW2025-34

[37] Michael Stonebraker, Randy H. Katz, David A. Patterson, and John K. Ousterhout. 1988. The Design of XPRS. In *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*. Morgan Kaufmann, 318–330. Retrieved June 29, 2026 from http://www.vldb.org/conf/1988/P318.PDF

[38] Chung-Piaw Teo, Jay Sethuraman, and Wee-Peng Tan. 2001. Gale-Shapley Stable Marriage Problem Revisited: Strategic Issues and Applications. *Manag. Sci.* 47, 9 (2001), 1252–1267. https://doi.org/10.1287/MNSC.47.9.1252.9784

[39] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. 2022. reformulator: Automated Refactoring of the N+1 Problem in Database-Backed Applications. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 84:1–84:12. https://doi.org/10.1145/3551349.3556911

[40] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. CWI/ACM, 2. Retrieved June 29, 2026 from http://event.cwi.nl/grades2013/02-xin.pdf