A Query Engine for the Agents

Kenny Daniel
Hyperparam
Seattle, USA
kenny@hyperparam.app

ABSTRACT

The fastest-growing data in production today is unstructured text: agent traces, chat logs, reasoning chains, model outputs. People want to analyze it, and the questions worth asking ("show me where the agent got confused") cannot be answered by SQL alone, since text is not queryable without a model in the query path. The natural place this analysis is happening is the new class of AI applications, Claude Code, Cursor, Claude Desktop, and in-browser agents, that run client-side and host both a human user and an LLM agent in the same process. These applications increasingly want to work with data, but the lakehouse read path has been hard to use from a JS runtime: Spark, Trino, and managed warehouses do not fit there. To build this new kind of AI data application, three properties of the engine become first-order: a JS-native distribution that drops into the runtime the application already runs in, a bundle small enough to ship inside a cold tab or per-turn agent sandbox, and a way to interleave analytic operators with model-based interpretation of text. We present **Hyperparam**, three open-source JavaScript libraries (Hyparquet, Squirreling, Icebird) totaling under 70 KB, that read Parquet and Apache Iceberg directly from object storage and meet the third property with per-cell, async-native SQL execution, so expensive cells fire only when downstream operators demand them. Squirreling runs LLM-shaped async UDFs over 300× faster than DuckDB-WASM on filter-bounded queries (and 192× on sort-bounded queries) and completes a ten-task agent analyst suite at two-thirds lower cost. We argue that data engineering as a discipline needs to update for the AI-native client applications now in production and the agents that work alongside their users.

CCS Concepts: Information systems → Data management systems; Computing methodologies → Intelligent agents.

Keywords: agentic data systems, lakehouse, Parquet, Iceberg, browser-native, late materialization, LLM-as-judge, agent traces

1 INTRODUCTION

The data that practitioners and agents most want to query is unstructured text, written by software: chat logs [8], tool-use traces [7], reasoning chains, coding-agent logs, and model outputs, at millions to billions of records per tenant [19]. Liu et al. [12] project that AI agents will become the dominant consumer of this data, issuing orders of magnitude more probes than humans; they report that, at Neon, agents created 20× more database branches and performed 50× more rollbacks relative to humans.

Classic OLAP was not built for this shape of data. The questions that surface real problems, like where users got frustrated, why a tool call failed, or where an agent went down a rabbit hole, cannot be answered by SQL alone [9]; they require a model in the query to interpret each row, with `llm()` UDFs as first-class operators alongside scans, joins, and aggregates, and cheaply, since each call is a billable inference. The data is heavy and its owners want it to stay where it already lives [5]. The conventional read path puts heavy infrastructure (Spark, Trino, a managed warehouse, a query gateway) between consumer and data, slowing the curate-review-act loop [2].

The natural place this analysis is happening is a new class of AI applications that has reached production. Claude Code (Bun CLI), Claude Desktop, Cursor, and VS Code (host to Copilot, Cline, and Continue) ship as Electron; hyperparam.app runs in a browser tab; per-turn agent sandboxes are Node `vm` contexts, Deno, or Web Workers. They share three properties: client-side, often JavaScript, and a process that hosts both a human user and an LLM agent. They increasingly want to query data, but the lakehouse read path does not fit a JS runtime: Spark, Trino, and managed warehouses are server-side. An agent issuing exploratory probes inside a tool-use turn cannot wait on a credentialed backend on the critical path either. The engine has to come to the consumer.

Our position. Three properties of the engine then become first-order: a JS-native distribution, so the engine drops into the runtime the application already runs in with no second runtime or FFI between user and agent; a bundle small enough to load inside a cold tab or a per-turn agent sandbox; and a way to interleave analytic operators with model-based interpretation of text, since trace cells can be tens of kilobytes of compressed text and an `llm()` UDF call is a multi-second remote inference with real dollar cost. Hyperparam meets the third with per-cell, async-native execution, where expensive cells fire only when downstream operators demand them and the query path can `await` model APIs as a first-class step. We ground the position in three open-source JavaScript libraries (Hyparquet, Squirreling, Icebird) totaling under 70 KB; DuckDB-WASM [10] is the closest prior art and we contrast in §2.

2 RELATED WORK

Lakehouse foundations. The lakehouse architecture [5] places a transactional metadata layer such as Apache Iceberg, Delta Lake, or Hudi on object storage, typically over Parquet [20]. Hyperparam reads both layers directly; the storage substrate is present in every production deployment we target.

In-place and embedded analytics. DuckDB [16] is an embeddable OLAP engine designed to run inside a host process; DuckDB-WASM [10] compiles it to WebAssembly and was ahead of the curve in establishing that interactive analytics belong in the browser. It is the prior art closest in spirit to Hyperparam, and we extend the thesis into the unstructured-text and agent regime on two axes. First, bundle size. DuckDB-WASM's core WebAssembly module is roughly 8 MB gzipped over the wire (35 MB uncompressed), plus a worker shim and JS glue; Hyparquet, Squirreling, and Icebird together ship in under 70 KB gzipped, a roughly 100× gap on the same axis that matters for cold-tab page loads from incident links and for

per-turn spin-up inside an agent sandbox. Second, the unit of laziness. In the DuckDB-WASM path we test, execution is vectorized in morsels, but scalar UDF calls still cross the WASM boundary synchronously from the engine's perspective: a long-running awaited call inside a UDF such as an `llm()` inference stalls that path rather than letting other rows continue around it. Squirreling's AsyncGenerator operators stream *deferred cells* throughout, so any cell can be an awaited remote or inference call, and cells that no downstream operator reads are never decompressed (§3.2). DuckDB-WASM remains the right tool for many workloads; the agentic-text loop is where the gap is widest.

Other JavaScript Parquet tooling does not address this workload. Arrow-JS [3] is oriented around Arrow IPC buffers and does not do HTTP-range reads over remote Parquet, `parquetjs` is unmaintained, DataFusion lacks a production-ready WASM package, and Arquero [14] operates over in-memory DataFrames and complements a storage access layer rather than replacing one.

Agentic data systems. Liu et al. [12] argue that the shift from human operators to AI agents fundamentally breaks existing data system assumptions, since agents issue orders of magnitude more probes, most redundant and speculative, and the query layer must become satisficing-aware. Tagliabue et al. [18] argue from the write side, where the agentic lakehouse needs transaction isolation and governance primitives for concurrent agent writes. Zaharia et al. [21] motivate the broader shift from monolithic models to compound AI systems. A parallel thread treats LLM-based operators as first-class query primitives, with accuracy guarantees [15], cost-aware optimization over unstructured collections [11], and dedicated engines for semantic predicates [6]. Hyperparam addresses the read seam these framings leave open. How does a user, or an agent, get to the data once it has been stored?

Trace observability platforms. Langfuse, Braintrust, LangSmith, Helicone, and Phoenix address the same user-facing workload by holding traces in a store they operate, on top of which they build curation and evaluation UIs. Hyperparam's stance is different: trace data stays with the owner in their existing storage, and the same JavaScript engine runs in whatever runtime the consumer is already in (browser tab, agent sandbox, or one tab hosting both), so the iteration loop closes inside one engine rather than being split between a vendor UI for the human and a hosted API for the agent.

3 THE HYPERPARAM STACK

Hyperparam is a browser-native lakehouse client built around Squirreling, a small async SQL engine with pluggable backends. The client issues SQL to Squirreling, which dispatches scans to whichever backend matches the source: Hyparquet for Parquet files, Icebird for Iceberg tables. Row-oriented adapters are available for CSV, JSONL, or HTTP APIs, but we focus on the column-oriented backends, as they offer the most performance benefits, and are widely used for large-scale storage. Each library is independently deployable, and together they turn a URL plus object-storage credentials into a live query target. Figure~1 shows the composition.

3.1 Hyparquet: Range-Efficient Parquet Reads

Hyparquet reads the Parquet footer via a speculative trailing range request (a HEAD-derived absolute byte offset, since browser CORS blocks the suffix-range form `Range: bytes=-N`), decodes the metadata, and issues parallel range requests for exactly the column chunks needed by the query, skipping row groups via min/max statistics. No full-file download occurs. At 14 KB gzipped with zero dependencies, it implements the full Parquet specification including nested types via Dremel-encoded repetition and definition levels [13], which matters for deeply nested LLM trace schemas. The practical effect on the consumer is fast spin-up: an incident link can open a 40 GB Iceberg table of agent traces on S3 and return the first row in under a second from a cold tab, with no `pip install`, cluster, or query gateway in the path. The cold-start gap is not our main claim, since most enterprise Trino and Spark deployments are pre-provisioned; the more durable comparison is warm-start, where each cluster probe still incurs a coordinator round trip and worker-side scan that the in-browser path avoids (Table 1).

Stack	Cold	Warm
Hyperparam	0.6 s	0.2 s
DuckDB-WASM	19 s	1.3 s
Trino on EC2	~3 min	1.1 s
Spark on EMR	~8 min	3.4 s

Table 1: time-to-first-row for `SELECT * FROM traces WHERE session_id = ? LIMIT 50` *on a 40 GB Iceberg table of agent traces on S3. "Cold" includes bundle fetch and engine init for browser stacks and cluster provision for Trino/Spark; "warm" assumes the engine is already up and footer/manifests are cached. DuckDB-WASM 1.33.1-dev45.0. Means across repeated trials.*

3.2 Squirreling: Async-Native SQL with Per-Cell Laziness

Squirreling is a from-scratch JavaScript SQL engine, 22 KB minified and gzipped, with zero dependencies, whose execution primitive is a *lazy deferred cell.* A cell encapsulates a computation such as a column decompression, a network fetch, or an LLM inference call, and is not evaluated until its value is demanded by a downstream operator or a `LIMIT` boundary. Its SQL dialect is deliberately permissive: the target consumer is a model rather than a fixed API contract, so the parser accepts aliases and idioms from multiple SQL dialects. Joins can span heterogeneous backends, e.g. an Iceberg table against a JSONL file in one query.

Squirreling's cell-level pull aligns cost with demand: a column chunk is not fetched until some row needs it, a value is decompressed only for rows that pass the predicate and reach a projecting operator, and an `llm()` call is not issued until the row reaches the output. This extends the late-materialization insight of Abadi et al. [1] beyond the column-store bandwidth argument to network I/O and LLM cost.

A query `SELECT llm('classify', content) FROM traces WHERE session_id = $id LIMIT 5` therefore makes exactly five inference calls, regardless of how many rows match the predicate before the limit. This guarantee holds on the streaming path taken by queries without ORDER BY, GROUP BY, or aggregates; queries that require a global sort or group switch to a buffered path, at the cost of the per-cell bound. On the streaming path, satisficing [12]

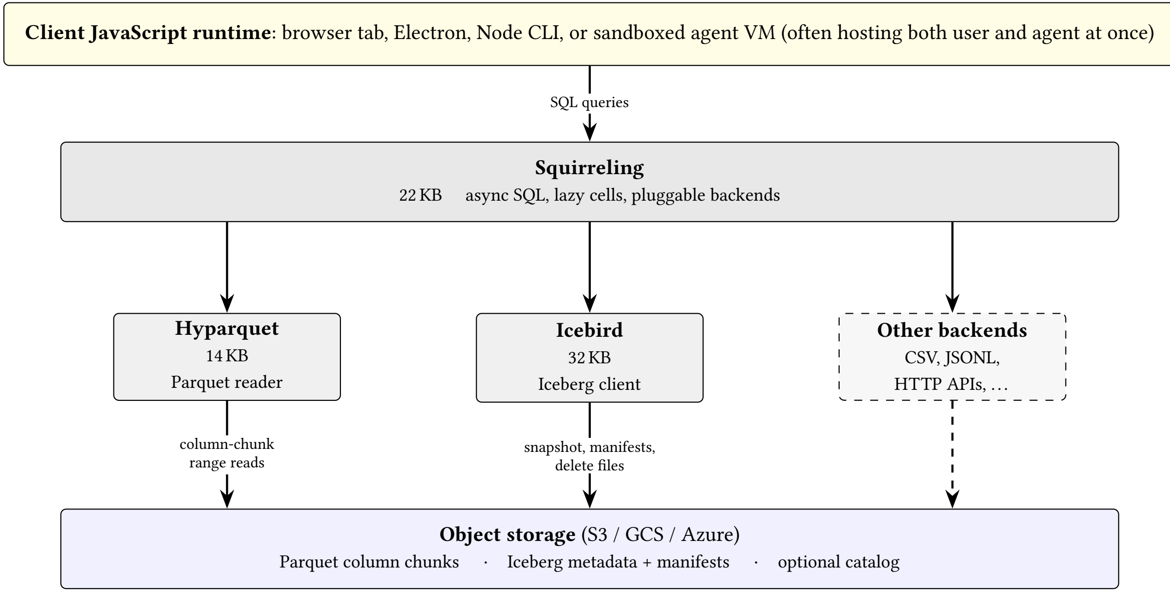

Figure 1: The Hyperparam stack: a client JS runtime issues SQL to Squirreling, which dispatches reads through pluggable backends (Hyparquet for Parquet, Icebird for Iceberg, others for CSV / JSONL / HTTP APIs) that fetch directly from object storage via a custom-fetch hook supplying credentials. Under 70 KB combined, gzipped. No backend service mediates between consumer and data.

becomes a query-time primitive. The consumer stops when enough rows have been seen, and the system charges only for what was read.

The async-native execution boundary is the other key property. Squirreling's operators are composed as JavaScript AsyncGenerators throughout. Every operator yields rows as they become available, every blocking operation is an awaited Promise, and the planner reasons over the dependency graph of deferred cells without a thread-pool abstraction. The `llm()` UDF accepts a model identifier, a prompt template, and column references; results stream back as they arrive, partial results are visible before completion, and the consumer can cancel mid-query. Human judgment stays in the loop, since Shankar et al. [17] show that LLM judges diverge systematically from human preference, so the UDF is a filter that surfaces rows worth reviewing, not a replacement.

We measure the practical effect in a head-to-head with DuckDB-WASM, the closest browser-native baseline (Table 2). The benchmark uses a mock `llm()` UDF with a 5 ms per-call delay, which is much faster than real hosted LLM inference; the point is to isolate call counts and async pipelining without making the experiment slow to run. Two independent properties drive the gap. *Plan-shape laziness* governs call count: on shape B, Squirreling's cell-level streaming short-circuits at the LIMIT (606 calls at N=50, 12.5% selectivity), while DuckDB-WASM cannot push LIMIT past a filter on a generated column and evaluates in 2,048-row morsels. On shape C, both engines must evaluate every row before the sort bounds the output, and call counts converge. *Async pipelining* governs wall-clock independent of call count: Squirreling's AsyncGenerator operators overlap up to 256 concurrent invocations, while DuckDB-WASM's `createScalarFunction` is a synchronous C++ call into WASM and pins concurrency at one. Registering an `async` callback does not recover it; DuckDB-WASM reads the unresolved Promise as the declared Arrow type and writes a zero into every result cell, so async UDFs return silently wrong answers rather than slow correct ones. On shape C, call-count parity still leaves a 192× wall-clock gap purely from pipelining.

Shape	DuckDB-WASM	Squirreling
A: `LIMIT N`	50 calls 320 ms	50 calls 20 ms
B: `WHERE llm(...)=1 LIMIT N`	2,048 calls 12,620 ms	606 calls 40 ms
C: `ORDER BY llm(...) LIMIT N`	10,050 calls 61,667 ms	10,000 calls 321 ms
D: `WHERE native LIMIT N`	50 calls 333 ms	50 calls 24 ms

Table 2: UDF call count and wall-clock at N=50 on a 10,000-row input with a 5 ms per-call mock latency, across four query shapes. Squirreling 0.12.16; DuckDB-WASM 1.33.1-dev45.0. Call counts are deterministic given the input; wall-clock is the median across 5 trials.

3.3 Icebird: Client-Side Iceberg

Icebird [4] resolves Iceberg snapshots client-side into the set of Parquet files Hyparquet then reads, applying position and equality deletes in the client. The central design decision is the custom-fetch hook. Callers inject a `fetch`-compatible function that supplies credentials such as AWS SigV4, GCS HMAC, or bearer tokens per request, so the engine itself stays credential-agnostic and

works against any object store the caller can already authenticate against. Icebird supports two access modes. If the caller already has a metadata-file or snapshot URI, Icebird can read it directly from object storage. If the caller needs current-table resolution, named branches, or stronger coordination under concurrent writers, Icebird uses an Iceberg catalog. Either way, the user and the agent each need only the table reference and the credentials they already use for storage or catalog access, with no shared backend on the hot path.

4 AGENTS AS CONSUMERS

Agents are the other major consumer of this trace data. Squirreling loads directly into whatever JavaScript runtime the agent is already executing in, whether that is Codex CLI's Node process, an ephemeral `vm.Context` spawned per agent in a swarm, or a browser tab. No separate query service sits in the path. For contrast, a Python query engine behind an agent tool typically means a containerized service per agent, with the per-turn cost of keeping it warm or paying its cold-start. Squirreling instantiates in milliseconds, and JavaScript's mature sandboxing makes per-agent isolation cheap. A production deployment runs at hyperparam.app, where investigation turns issue 10 to 50 probes against the trace table (§5).

We test the agent-consumer case directly in a head-to-head with DuckDB-WASM on the same Parquet corpus used in §3.2. An Anthropic Haiku-class agent is handed an identical `run_sql` tool backed by each engine and asked to answer ten analyst-style questions about a 50,000-row agent-trace dataset, phrased the way a practitioner would ask them, like "which session had the roughest ride?" or "on turns where the `web_search` tool is invoked, how often does the turn end in an error?", rather than as direct SQL translations. Across 5 independent trials per task (50 runs per stack), both stacks resolve all ten tasks correctly, but Squirreling does so at roughly a third of DuckDB-WASM's mean cost per ten-task pass (Table 3).

Stack	Correct	Median rounds	Cost per pass
DuckDB-WASM	50 / 50	2	\$0.203 (\$0.10–\$0.28)
Squirreling	50 / 50	2	\$0.067 (\$0.05–\$0.08)

Table 3: Anthropic Haiku agent completing a ten-task analyst-style suite over a 50,000-row agent-trace Parquet with `run_sql` backed by each browser engine, 5 trials per task. Squirreling 0.12.16; DuckDB-WASM 1.33.1-dev45.0. Round cap R=10 with up to 3 additional inferences for the agent to submit an answer. Correctness is scored against a reference answer pinned per task, with per-task numeric tolerance and case-insensitive string matching. "Cost per pass" is mean Anthropic Haiku list pricing summed across all tool-use rounds for one ten-task pass, with min–max across the 5 passes in parentheses.

The cost gap is almost entirely input tokens: across the 50 runs, DuckDB-WASM consumed 3.67× the input tokens Squirreling did (876K vs 239K) against only 1.43× the output (28K vs 19K). The agent emits roughly the same volume of SQL and prose on both stacks; what differs is what the model is *billed to read* on each subsequent call. The Anthropic API bills `input_tokens` for the full conversation context on every `messages.create`, so each prior round's `tool_result` is re-billed on every later inference, and total input grows quadratically in round count. Round count, in turn, is set by the error surface. Squirreling's "did you mean" hints on near-miss function names (e.g. `lenght` → `length`) and binder errors that enumerate available columns let the agent commit to a working query sooner: on T7 ("% turns that invoke a tool") DuckDB-WASM took a median 6 rounds and 42K mean input tokens vs Squirreling's 1 round and 2.4K; on T4 ("`web_search` turn error rate"), 7 rounds and 72K vs 3 rounds and 7.1K. Steerability [12] at the error surface is the proximate cause, and it compounds quadratically into the input-token bill.

Prompt caching narrows but does not close the gap: Anthropic `cache_control` breakpoints at tool-result boundaries take prior-context tokens to 10% of full price, dropping the R=5-vs-R=2 input-token ratio from 5× to 2.86×, but output tokens are still billed at full rate and the round-count incentive remains.

5 EXPERIENCE AND LIMITATIONS

Production deployment. Each library ships with a live public demo, reproducible against multi-gigabyte Parquet on Hugging Face or Iceberg snapshots in S3. The stack is also used in production by Hyperparam's own product at https://hyperparam.app, where in-browser agents import Hyparquet and Squirreling into the user's runtime; a typical investigation turn issues 10 to 50 probes against the trace table, and subagent orchestration pushes that higher. Hyperparam Desktop, an Electron app, adds persistent authentication for private S3 buckets and Iceberg catalogs behind corporate auth, resolving the credential-refresh and CORS constraints for the private-data case without giving up the in-place model. These deployments shaped the per-cell laziness design, since agent turns demand tight LLM-cost bounds, and the custom-fetch hook in Icebird, since credentials never leave the caller's execution context.

Where backend-free is the wrong choice. Authentication against private object stores is a real client-side concern, and the right answer depends on the host runtime. A web tab needs short-lived browser-vended tokens, an Electron app can persist credentials in the OS keychain, which is the route Hyperparam Desktop takes for private S3, and a Node CLI inherits whatever the shell already has. Aggregations over tens of billions of rows, multi-table joins exceeding RAM, or workloads that need server-side spill-to-disk are outside the operating envelope; the terabyte join still belongs on a cluster. Per-cell laziness bounds inference cost for `LIMIT`-bounded queries but does not prevent an unbounded `SELECT llm(...) FROM traces`; rate limiting, cost preview, and per-UDF token budgets are open UX problems.

6 CONCLUSION

AI applications run on the client now, in JavaScript, with agents and humans sharing one process and reasoning over text data their owners do not want to move. The data engineering stack, built for a server-side, SQL-only, human-paced era, sits on the wrong side of the runtime boundary for this class of application. We argue that data engineering as a discipline needs to update for this new shape of data and the new shape of queries that run on it.

REFERENCES

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*.

[2] Sara Alspaugh, Beidi Chen, Jessica Lin, Archana Ganapathi, Marti A. Hearst, and Randy Katz. 2014. Analyzing Log Analysis: An Empirical Study of User Log Mining. In *LISA*. USENIX Association.

[3] Apache Arrow. 2024. Apache Arrow JavaScript Library (arrow-js). https://arrow.apache.org/docs/js/

[4] Apache Iceberg. 2024. Apache Iceberg Table Spec v2. https://iceberg.apache.org/spec/

[5] Michael Armbrust, Ali Ghodsi, Reynold S. Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms That Unify Data Warehousing and Advanced Analytics. In *CIDR*.

[6] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. *arXiv:2407.09522* (2024).

[7] Liming Dong, Qinghua Lu, and Liming Zhu. 2024. AgentOps: Enabling Observability of LLM Agents. *arXiv:2411.05285* (2024).

[8] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2021. A Survey on Automated Log Analysis for Reliability Engineering. *Comput. Surveys* 54, 6, Article 130 (2021). https://doi.org/10.1145/3460345

[9] Rogers Jeffrey Leo John, Dylan Bacon, Junda Chen, Ushmal Ramesh, Jiatong Li, Deepan Das, Robert Claus, Amos Kendall, and Jignesh M. Patel. 2023. DataChat: An Intuitive and Collaborative Data Analytics Platform. In *SIGMOD Companion*. https://doi.org/10.1145/3555041.3589678

[10] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. 2022. DuckDB-Wasm: Fast Analytical Processing for the Web. *Proceedings of the VLDB Endowment* 15, 12 (2022).

[11] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Bailis Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*.

[12] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. *arXiv:2509.00997* (2025).

[13] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment* 3, 1 (2010).

[14] Dominik Moritz and Jeffrey Heer. 2020. Arquero: Query Processing and Transformation of Array-Backed Data Tables. Observable. https://observablehq.com/@uwdata/arquero

[15] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* (2025). https://doi.org/10.14778/3749646.3749685

[16] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*.

[17] Shreya Shankar, J.D. Zamfirescu-Pereira, Björn Hartmann, Aditya G. Parameswaran, and Ian Arawjo. 2024. Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences. In *UIST*.

[18] Jacopo Tagliabue, Federico Bianchi, and Ciro Greco. 2025. Trustworthy AI in the Agentic Lakehouse: from Concurrency to Governance. *arXiv:2511.16402* (2025).

[19] The Deep View. 2026. AI's compute crisis has reached a breaking point. https://www.thedeepview.com/articles/ai-s-compute-crisis-has-reached-a-breaking-point Accessed 2026.

[20] Deepak Vohra. 2016. Apache Parquet. In *Practical Hadoop Ecosystem*. Apress.

[21] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. BAIR Blog. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/