Bulletin of the Technical Community on

Data Engineering

March 2026 Vol. 50 No. 1 IEEE Computer Society

Letters

Special Issue on LLM-Powered Data Processing

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Community on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modeling, theory and application of database systems and technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of TC on Data Engineering, IEEE Computer Society, or authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

For decades, the relational model and declarative query languages have served as the North Star of our community. We built massive, general-purpose engines based on a simple, powerful philosophy: users declare what data they want, and the system figures out how to retrieve it. Today, we stand at the precipice of a foundational paradigm shift that elevates this abstraction even further. We are transitioning from declarative data management to agentic data management. In this new era, we are no longer just asking our systems to passively fetch records; we are asking autonomous agents to navigate ambiguity, synthesize multimodal evidence, and actively mutate our enterprise environments to achieve high-level business objectives.

The papers in this special issue capture the sheer magnitude of this pivot, separating the rigid architectures that will fall away from the bedrock principles that will endure. Perhaps the most striking change is the shift in how databases are constructed. The long-standing pursuit of a single, general-purpose engine designed to handle any arbitrary schema or workload is being challenged. Eckmann et al. present a bold vision where the future is bespoke. They propose leveraging large language models to automatically synthesize full database systems on demand, crafting "one-size-fits-one" engines that shed the overhead of general-purpose flexibility to hardcode data structures and algorithms optimal for a single specific workload.

Simultaneously, the role of the database is shifting from a passive system of record to a high-stakes, active nervous system. As Ang et al. note, modern data agents are transitioning from simply reading data to acting on it, bringing real-world consequences and coupling mutation with profound risk. To safely unleash these capabilities, our static databases must evolve into Agentic Data Environments that actively discover information while natively supporting speculative branching and enforcing strict data flow controls across the entire computing stack. This underscores a broader reality explored by Özsu et al.: there is a powerful duality between LLMs and data management systems. While LLMs are revolutionizing natural language querying and data integration, our underlying data architectures must be fundamentally redesigned to support the unpredictable, multi-step workloads required to train and serve these models efficiently.

Yet, as we embrace these autonomous, bespoke, and active environments, the foundational principles of our discipline are not dying—they are reaching their ultimate form. The declarative dream is evolving into true model-data independence. Patel et al. introduce the semantic operator model, which allows programmers to write application logic using high-level, natural language operations while the system transparently optimizes the underlying, complex AI executions. To make these semantic operations financially and computationally viable, we must still rely on classic, hardcore database engineering. Trummer demonstrates this beautifully by reimagining the traditional block nested loops join for semantic operations, proving that we can optimize batch sizes to dramatically reduce the costs and API overhead of LLM invocations.

Above all, the agentic future reaffirms that advanced reasoning is entirely useless—and often dangerous—without a rigorous grounding in reality. Chung et al. emphasize that embedding AI capabilities directly into the core engine is only part of the solution; the reliability of these autonomous systems is strictly predicated on a rigorous semantic foundation. Universal metadata catalogs, schema lineage, and robust semantic models will remain the indispensable anchors that keep non-deterministic LLMs tethered to deterministic enterprise facts.

The shift from declarative to agentic data management is ultimately about relinquishing micro-management in exchange for goal-oriented autonomy, while simultaneously building stronger, smarter architectures to govern that autonomy. I invite you to read the outstanding contributions in this issue

and join us in building this bespoke, agentic future.

Haixun Wang
EvenUp

Letter from the Special Issue Editor

The integration of Large Language Models (LLMs) into data systems marks a fundamental transition toward AI-native environments, where reasoning is elevated to a first-class primitive within the query-processing engine. Rather than serving as external tools, LLMs introduce intrinsic capabilities for autonomous semantic synthesis, multi-step logical inference, and the systematic processing of unstructured data. This shift allows modern data architectures to provide the same analytical rigor for heterogeneous content that was previously restricted to structured data. As autonomous agents and generative workflows redefine computing requirements, the data stack is being re-engineered to facilitate these high-level cognitive tasks at scale. This special issue examines the technical frontiers of LLM-powered data processing, featuring six articles that investigate the evolution from traditional architectures toward a new generation of agentic, reasoning-capable data environments.

Building on the need for agent-friendly systems, Ang et al. introduce the concept of **Agentic Data Environments**. They highlight the critical transition from read-only AI copilots to read-write autonomous agents capable of mutating environment state. To support these agents safely, they propose infrastructure that actively curates task-relevant signals (via Agentic Information Management and Retrieval) while bounding catastrophic risks through advanced mechanisms like state branching and deterministic Data Flow Control (DFC).

Grounding the architectural shifts in the enterprise, Chung et al. present Google Cloud's transition to an **Agentic Data Cloud**, enabling intelligent and autonomous agents and applications. They describe several building blocks of their scalable AI Agentic Data Cloud ecosystem, designed to support production-scale autonomous applications and agents, including AI functions which are built-in directly into the core processing engines, natural language interfaces to data, and a modern search stack. Crucially, they assert that the trustworthiness of these non-deterministic agents relies heavily on a rigorous semantic foundation powered by universal metadata catalogs and rich semantic models.

The Özsu et al. claim that the traditional approach of treating AI as an auxiliary service layered over isolated databases is no longer sufficient. They frame this transition as a profound **"duality" (LLM4DB and DB4LLM)**, arguing that current data systems are ill-equipped for LLM agents that exhibit variable execution times, non-deterministic behaviors, and ambiguous write operations. To address complex analytical tasks over heterogeneous federated systems, they propose a novel multi-agent architecture where Specialized Data Agents (SDAs) interact natively with individual sources, coordinated by a central orchestrator ensuring strict data provenance.

Patel et al. introduce the **Semantic Operator Model**, implemented in the open-source LOTUS system. By introducing "model-data independence," this framework allows developers to write high-level natural language queries , such as sem_filter, sem_join, and sem_topk, while the underlying system transparently optimizes execution. The system explores cost-reducing proxies and model cascades while providing strict statistical accuracy guarantees relative to a high-quality reference algorithm.

Focusing on the algorithmic execution of the semantic operators, Trummer tackles the high costs of language-based joins with **Optimal Block Nested Loops Implementations for Semantic Joins**. Traditional semantic processing often relies on naive tuple-to-tuple comparisons, resulting in quadratic LLM invocation costs. Trummer proposes a block nested loops approach that batches multiple rows into a single prompt, providing a formal cost model to determine the optimal batch sizes within the strict constraints of limited LLM context windows.

Finally, in the last paper, **The Future Is Bespoke**, Eckmann et al. challenge the decades-old "One Size Fits All" philosophy of general-purpose engines. Instead, they propose utilizing LLM coding agents to automatically synthesize "one-size-fits-one" DBMSs tailored entirely to a specific workload. By shedding the overhead of schema interpretation, generalized data structures, and unused code paths, their synthesized bespoke engines demonstrate staggering performance improvements—achieving up to

an 11.78x speedup over state-of-the-art general-purpose engines like DuckDB.

Together, these six articles present a cohesive vision of the future. Whether it is through semantic operators, specialized hardware-software synthesis, or robust multi-agent orchestration, the boundary between the database and the artificial intelligence that consumes it is rapidly dissolving. We hope this special issue inspires further research into the foundations of AI-native data systems.

We would like to thank all the authors for their valuable contributions. We also thank Haixun Wang for the opportunity to put together this special issue, and Jieming Shi for his help in its publication.

Fatma Özcan
Systems Research, Google

Agentic Data Environments

Elaine Ang, Chenxi Huang, Georgios Liargkovas, Jerry Liu, Jinhui Liu, Nikos Pagonas,
Charlie Summers, Haonan Wang, Jiakai Xu, Tianle Zhou, Yusen Zhang,
Zhou Yu, Zhuo Zhang, Tianyi Peng, Kostis Kaffes, Eugene Wu
DAPLab, Columbia University
{ra3448, ch4023, gl2902, jl6235, jl7309, np2948, cgs2161, hw2983, ax2155, mz2998, yz5296,
zy2461, zz3474, tp2845}@columbia.edu, {kkaffes, ewu}@cs.columbia.edu

1 Introduction

Automation has long been the promise of computing. The introduction of modern large language models [1] (LLMs) has changed who (or what) performs this automation. LLMs, combined with vibe coding, agent frameworks, and rich API ecosystems, empowered non-programmers to deploy autonomous agents that operate terminals [2], call APIs and tools [3–5], navigate GUIs [6], code [7], and query databases [8, 9]. Rather than copilots that recommend actions for the user, agents autonomously observe data, plan and execute actions, and observe their *effects. This shift from reading data to acting on it is the central challenge in future data management.*

Today's data agents are largely read-only. NL2SQL, retrieval-augmented question answering, and data analytics agents observe data, synthesize it, and return an answer. A tax reporting agent may retrieve financial statements and transaction records to estimate last quarter's revenue; its actions make no long-term side effects to the environment. This design simplifies evaluation, improves failure tolerance, and limits potential harm.

In contrast, agentic automation mutates the environment with real consequences. The same tax scenario is fundamentally different when the agent also reconciles discrepancies across financial statements, applies tax logic, and files official returns. Each step is simultaneously a data write and a consequential action that e.g., modifies accounting records, overwrites prior filings, and submits legally binding documents. Because mutation and consequence are coupled, errors are not merely wrong answers: they can lead to regulatory penalties, lawsuits, or compliance violations. Agentic automation is ultimately a read-write problem: when agents can modify data, the value of automation shifts from what agents can accomplish to what happens when they fail.

1.1 Automation's Value Proposition

To make this trade-off precise, consider the core value proposition for agentic automation:

$$\text{Value} = \text{Benefits} - \text{Costs} \tag{1}$$

Automation promises substantial benefits through speed, scale, and labor savings. However, the cost of failure differs in character and magnitude. Benefits accumulate gradually across many successes, but costs are abrupt, catastrophic, and difficult to reverse: deleting a production database [10], triggering a cloud outage [11], and exfiltrating data [12–15]. Because agents operate over systems of record, failures can propagate before detection. In both perception and practice, the potential costs of agent automation therefore appear unbounded.

This asymmetry shapes adoption. Users do not calibrate trust based on overall performance, and a single salient failure can suppress adoption out of proportion to its likelihood [16]. This is corroborated by prospect theory, which finds that humans weigh losses much more heavily than the same gains [17]. As a result, those evaluating automation focus on worst-case outcomes rather than expected performance, and systems that are only safe in the common case remain unsuitable for important tasks.

The implication is that "best-effort" safety is not enough, as higher average reliability does not affect adoption if catastrophic outcomes remain plausible. We need to both increase the benefits of automation *and* bound the consequences of failure. This is not solely an agent design problem, but a systems problem: the environment that the agent executes within and the guarantees the environment provides.

1.2 From Databases to Data Environments

Databases remain a central component of modern computing. However, automation goes well beyond simply querying databases or perform analytics. Real-world tasks require interacting with the broader computing stack, including applications, APIs, files, configuration systems, command-line tools, and external services.

Example 1: *An agent send an HTTP request to a web service that triggers server-side business logic, which launches background jobs, calls external APIs, writes to the file system, and mutates the database. Through this chain, the agent indirectly interacts with multiple subsystems and a considerable amount of evolving state: application variables in the server process, configuration files, job queues and logs produced by background tasks, files written to disk, responses from external services, and persistent records in the database. The outcome and effects depend on database contents as well as the configuration and state of the surrounding environment.*

This suggests that "data management" for agentic automation must extend beyond databases to the *data environment*: the collection of heterogeneous resources that the agent runs and interacts within—including data lakes, file systems, memory, APIs, derived artifacts, processes, and system metadata—along with the mechanisms that govern how this state is accessed, modified, and allowed to flow. Unlike a classic Database Management System (DBMS), data environments encompass a broad range of data models and software components rather than a single data store. Unlike a dataspace [18], which focuses on integrating heterogeneous data sources, data environments are the *stateful substrate* that the agent executes within.

1.3 Towards Agentic Data Environments

What does a data environment designed for agents rather than humans look like?

Modern computing systems can already be viewed as a form of *data environment*. Filesystems, databases, services, and APIs collectively form a shared state space within which programs operate. However, these environments are largely *passive*: they serve data and compute requests, but do not actively organize information, guide decision making, or constrain data flows and data is used.

Agentic automation demands more. Agents continuously observe, act, and adapt to the data environment, and are capable of reasoning over

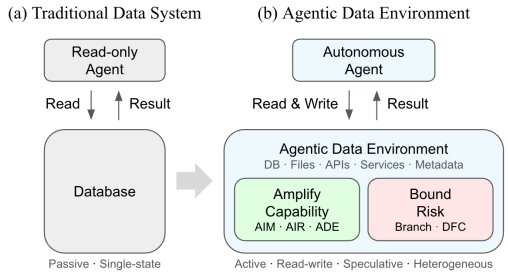


Figure 1: This paper describes a shift from systems for today's analytic agents to Agentic Data Environments.

large amounts of technical content far beyond any human. Yet, agents are ultimately consumers and producers of data: its decisions depend on the quality of the information it uses and its ability to safely explore its action space. Thus, supporting agents requires the data environment itself to become more than a passive store of state. These ***Agentic Data Environments*** actively prepare information and govern agent interactions in order to support reliable and safe automation. Their role is to:

- **Amplify Capability.** the environment must actively discover, materialise, organize and expose information from heterogeneous sources—documents, databases, data lakes, APIs, and the system itself—in forms that empower the agent without requiring developers to manually engineer every data pipeline.
- **Bound Risk.** the environment must provide guarantees that today's computing stacks lack: isolation and sandboxing to limit failures, transactional semantics and versioning across components, branching and rollback for speculative exploration, and policy enforcement to control how agents use and transform information.

This paper outlines the foundations of *Agentic Data Environments*. We describe three information management challenges based on how information becomes available: discovering agent-relevant data in massive data lakes, transforming data into agent-ready representations, and collecting/generating latent signals from the environment. We then discuss the infrastructure needed for safe agent exploration, including branching and data safety mechanisms. These same environment capabilities are also critical for training better agents (Section 4).

2 Agentic Data Environments To Improve Agent Capabilities

To increase the benefits of automation, agents must access and reason over the right information. Agent failures are increasingly information rather than reasoning failures, and even the most powerful models cannot solve tasks when the relevant signals are missing, poorly structured, or undiscoverable.

The responsibility of the data environment is to manage, find, elicit, and deliver the right information, in the right representation, at the right time, for the right task.

This need is particularly acute for a new class of agent builders: domain experts who can vibe code agents without formal software development training. While they can identify relevant data sources or share domain knowledge, turning that information into agent-ready representations remains out of reach. Ultimately, the challenge is developer ergonomics: how to use domain expertise while hiding the complexity of data management?

The key observation is that the data consumer is no longer a human. Traditional data products aim to faithfully represent reality for analysts. Agents instead treat data as a means to an end: task success. Data environments must shift from representing reality toward preparing task-relevant signals for agents.

The main variable across settings is how available the required information already is. In some cases the relevant data source is known but poorly structured for the task. In others the information exists but must be discovered within a large heterogeneous data lake. Finally, the needed signal may exist only implicitly in the environment and must be inferred or materialized.

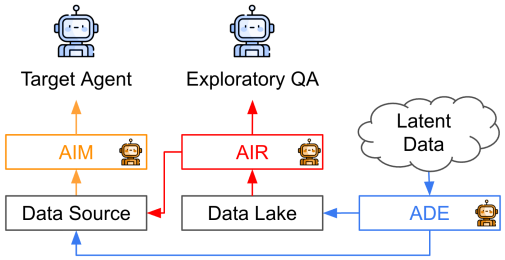


Figure 2: AIR finds relevant data sources for EQA or target agents, AIM turns sources into capabilities, and ADE materializes missing signals into new data sources. curates information to improve .

These settings motivate three complementary directions. *Agentic Information Management (AIM)* transforms known sources into agent-ready capabilities. *Agentic Information Retrieval (AIR)* finds sources with the needed evidence by a task. *Agentic Data Elicitation (ADE)* surfaces latent signals that are not yet materialized as artifacts.

2.1 Agentic Information Management: Taking AIM at Data

Today, the dominant approaches that make data accessible to agents will convert raw sources into vector embeddings for retrieval-augmented generation (RAG) [19], or store them in a default log system [20], document store [21], or text files [22]. Many refer to such processes as agent context or agent memory curation. These generic representations impose a fixed schema, ignorant of the agent's task, and lose the structure that may be critical for downstream reasoning. For instance, using RAG for a conversation corpus loses temporal ordering, speaker identity, and cross-session relationships needed by a question-answering agent.

Tasks, data, and models evolve over time, and static representations quickly grow stale. Thus, we propose ***Agentic Information Management (AIM)*** to automatically manage information representations to benefit the domain expert's target agent. Given a data source and high-level expert guidance, AIM proposes a data model, schema, and extraction pipeline that captures the structure it believes will best support the target agent's tasks.

Example 2: *LoCoMo [23] is a multi-session conversation dataset where two speakers (e.g., Caroline and Melanie) converse across 19 sessions over several months. The dialogues cover events, career plans, hobbies, family activities, and evolving personal interests. Rather than embed this corpus into a vector store, an AIM agent reads the dialogues and proposes a relational schema tailored for what it expects to need. For example, the agent may design tables for* `Users`*,* `Sessions`*,* `Messages`*,* `Events`*,* `Interests_Activities`*, and* `Relationships`*. It then extracts and loads the session facts (e.g., "Caroline attended an LGBTQ support group on May 7, 2023") into the database, and exposes a SQL skill to this database. At query time, the target agent simply writes SQL queries against this database. To answer "What hobby does Melanie use to relax?", it queries* `Interests_Activities` *filtered by activity type rather than filtering hundreds of raw dialogues.*

Concretely, AIM is a multi-agent system that progressively adds structure and task-specialization while preserving the ability to evolve (Figure 3). The *Learning* stage analyzes the data source and hypothesizes data models for the target use cases. *Schema Modeling* then instantiates schemas and constraints in a specific data system (e.g., RAG, RDBMS, etc). *Data Loading* generates an Extract-Transform-Load pipeline to load the data source into the data system, and *Refinement* performs physical design to generate views and indexes for fast access. The resulting database is exposed as a *Data-Oriented Tool* or *Skill* that the target agent can call to retrieve task-relevant information. Each stage is informed by the domain expert's high level guidance (e.g., "the last user should be useful") and can use an extensible set of tools that run atop

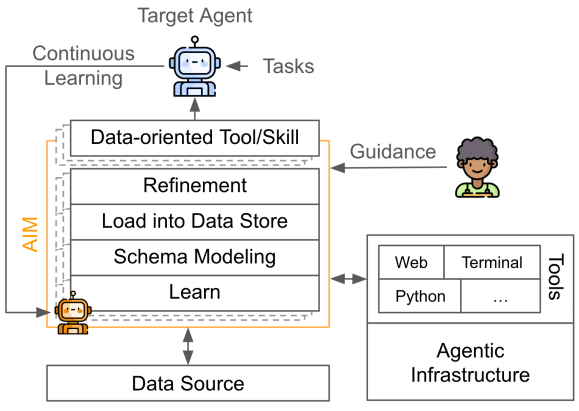


Figure 3: AIM takes a target agent, guidance, a set of tools, and a data source as input, and generates pipelines and artifacts to improve the target agent's task quality.

the Agentic Data Environment infrastructure (Section 3).The pipeline is agentically generated, so stages are revised based on new guidance, task feedback, and source changes.

Example 3: *On LoCoMo [23], AIM's accuracy is comparable to adding the full dialogue to the context of a frontier model, but uses only* ∼10% *the context length. When compared with specialized agent memory systems like Mem0 [20] and the SOTA RAG-based Octen [24], AIM is* 49.8% *and* 15.82% *more accurate, respectively. AIM is* 4.18× *faster than the state of the art agentic memory system GAM [25] and has* 13.54% *higher relative accuracy on average. In particular, AIM is* 11.53% *more accurate for temporal and* 24% *for open-domain reasoning questions because its structured representations are more effective than text file and RAG representations.*

2.1.1 Research Directions

The similarity between AIM and an automated data integration pipeline is intentional. The key difference is the objective: rather than create a high-quality schema upfront, AIM rapidly bootstraps an *agent-consumable capability* and evolves it over time. However, using agents to manage schemas and representations introduces challenges. There is no "correct" schema: the same conversation be modeled as an event timeline for one task and a relationship graph for another, and different extraction strategies can vary in performance. Further, the optimal representation is not static—models, prompts, and tools regularly change—so AIM must continuously "sample" for better pipelines (dashed gray boxes) while accounting for migration costs. Consequently, the data environment must be designed to rapidly generate, evaluate, and migrate pipelines; use task outcomes as feedback to improve or discard pipelines; and select the appropriate pipeline per request.

2.2 Agentic Information Retrieval: Breathing AIR into the Lake

AIM assumes that the data has been found. In practice, useful information is distributed across heterogeneous collections of millions of documents and datasets—a *data lake*—and the agent must first discover which sources contain the evidence needed to complete a task. We call this problem ***Agentic Information Retrieval*** **(AIR)**.

Data discovery systems seek to find datasets relevant to a query or example table. However, their evaluation measures—e.g., keyword matching, dataset similarity, joinability, or prediction accuracy—are only proxies for the downstream tasks users want. One common task is question answering (QA), where the goal is to retrieve evidence and synthesize an answer. QA spans reading comprehension [26–29], open-domain question answering [30, 31], and tabular question answering [32–34].

Recent LLM-based QA agents assume that the relevant evidence is in the context or easily found in a small curated corpus. In data lakes, however, the structure, semantics, and source relationships are largely unknown, and their scale is too large to fully analyze with LLMs. We call this setting ***Exploratory Question Answering (EQA)***: the agent must iteratively infer needed evidence and search the lake to find sources with the evidence.

To study this, we constructed LakeEQA, an EQA benchmark over a 9.5 TB data lake containing ∼40M documents from Wikipedia and Data.gov. Tasks must find and reason across multiple heterogeneous sources (7.67 documents on average) drawn from millions of candidates. Of particular importance is that tasks must *require search over the data lake.*

In many tasks, LLMs can easily hallucinate correct answers to individual steps or the entire task. To enforce this property, each step 1) must require accurate answers from previous steps to formulate its evidential need, and 2) the answer must be data dependent on a derived statistic or inferred fact from content in the lake. For example, identifying neighborhood schools that satisfy a class-size constraint

requires locating datasets describing school statistics and neighborhood boundaries, and reasoning across them sequentially.

To ensure that the 1000+ tasks are of high quality and do not contain annotation errors [35], the tasks are created by a team of 5 database Ph.D. students and 4 senior computer science undergraduates that passed our data science proficiency exam. Each task is validated by 4 annotators including at least one Ph.D.

LakeQA uniquely stresses two dimensions. *Search intensity* measures the difficulty to find relevant sources in the data lake, and *Reasoning intensity* measures the amount of multi-document inference required given those sources. As shown in Figure 4, existing benchmarks emphasize only one axis: multi-hop QA increases reasoning depth but provides the relevant documents, while dataset search emphasizes discovery but does not require downstream reasoning. In contrast, LakeQA requires both.

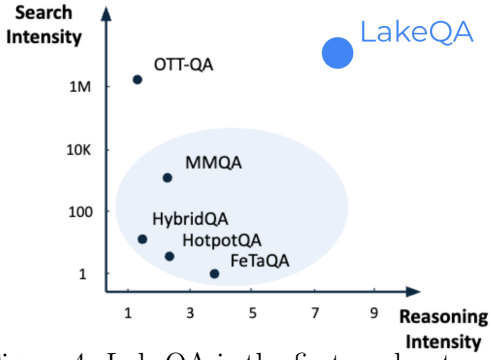


Figure 4: LakeQA is the first exploratory QA benchmark that pushes search intensity and reasoning intensity. Its 1000+ tasks are created and validated by 4 independent annotators including one database PhD.

Across seven frontier models, end-to-end accuracy is $\leq$23%. The dominant failure mode is failure to find the required datasets, not reasoning. This suggests that EQA agents must explore the data lake as part of the reasoning process; at this scale, the search system's design is as critical as the model's capability.

2.2.1 Research Directions

LakeQA shows that AIR must address two related challenges: represent the contents of a massive data lake so relevant sources can be discovered efficiently, and reason over how those sources can be combined to produce an answer. Discovery motivates a *semantic layer* between the agent and the data lake that summarizes what data exists, what it represents, and how sources may relate. Such summaries necessarily compress the lake. If compressed too aggressively, relevant sources become undiscoverable; if too coarse, the agent must evaluate large numbers of false positives that quickly exhaust latency, token, and cost budgets. Reasoning challenges arise when candidate sources are found. Answering a question must compose evidence across datasets by applying e.g., joins, entity resolution, temporal alignment, or reconciling conflicting definitions. The space of possible compositions is combinatorial, so AIR must jointly reason about *what sources to retrieve* and *how to compose them* to allocated limited compute and context to the most promising reasoning paths.

2.3 Agentic Data Elicitation: Distilling ADE from the Ether

The previous subsections assume that the relevant information already exists as data and must only be found or organized. In practice, however, many tasks depend on information that is only implicitly present in the environment. This *latent data* refers to signals that can be surfaced from the environment through observation, synthesis, or controlled experimentation. *Agentic Data Elicitation (ADE)* turns these implicit signals into explicit artifacts that future agents can reuse.

Latent data may capture semantic structure—such as implicit table roles, undocumented relationships between attributes, or derived business logic—or performance structure—such as workload regimes, bottleneck signatures, and relationships between control knobs and system metrics. The agent decides

what to inspect, hypotheses to test, experiments to run, and candidate artifacts to retain. The environment exposes the observation and control surfaces that make this possible—e.g., example, metadata, row samples, safe program execution, workload traces, metric probes, and actuation interfaces—and provides mechanisms to validate, store, and serve the resulting artifacts. Elicitation may be *passive*, when the agent collects existing state or history, or *active*, when it performs controlled interventions to reveal hidden response structure.

Example 4: *Consider a simple example from NL2SQL systems. A user asks: "How many sales activities does each account have?" The database contains tables* `ActivityHistory`, `Task`, *and* `Event`. *An agent that relies only on schema names may incorrectly treat* `ActivityHistory` *as the fact table. In practice, business activities are stored in* `Task` *and* `Event`, *while* `ActivityHistory` *records change logs.*

Through ADE—inspecting schema structure, sampling rows, or observing prior query traces—the agent can elicit the latent artifact "`ActivityHistory` *is a change-log table." Once materialized, this artifact becomes another AIM object that improves downstream reasoning such as table selection and query synthesis.*

Various agents interacting with systems already follow this pattern. The Tk-Boost [36] NL2SQL system materializes "tribal knowledge" by generating corrections from past interactions. REDSQL [37] materializes latent constraints (e.g., join consistency, valid aggregations) not in the schema; AgentSM [38] materializes example trajectories by synthesizing question sets and multi-step reasoning traces. This perspective extends from semantics to performance, where the objective is optimization rather than correctness. For example, to tune an OS scheduler [39], the agent must elicit how scheduler knobs affect performance by collecting runtime measurements, system state, and domain hints. Although their mechanisms differ, these approaches distill implicit structure from experience, constraints, or reasoning traces into explicit artifacts that persist beyond a single task.

Therefore, ADE-capable environments (e.g., OS, DBMS, system components) should expose APIs for agents to probe and learn from implicit signals. Beyond passive observation, environments should also provide safe mechanisms for agents to run controlled experiments (e.g., testing alternative indexes, cache policies, or operator strategies) while ensuring that changes remain sandboxed, auditable, and reversible. In a sense, ADE capabilities resemble eBPF-style extensibility [40, 41], where agents attach lightweight logic to system hooks to measure behavior, test hypotheses, and materialize reusable artifacts about system dynamics.

Many useful artifacts follow this pattern. In data-centric settings, agents may elicit statistical summaries, common join paths, derived metrics embedded in business logic, or schema groupings learned from past workloads. In systems settings, agents may elicit bottleneck signatures, knob-sensitivity maps, workload phases, or regime shifts. These artifacts improve performance not by changing the underlying model, but by making more of the environment's structure explicit and available to future agents.

2.3.1 Research Directions

This opportunity is to reason about latent structure and materialize it. Once elicited and validated, many artifacts—such as performance counters, inferred schema relationships, reusable query hints, or tuning knowledge—can be managed as standard AIM data sources and reused across tasks. To support this ADE-capability, environments must make latent structure easy to elicit, validate, persist, and refresh. This demands low-latency access to instrumentation and sampling interfaces, methods to validate potentially spurious or overfit candidate artifacts before they are reused, efficient ways to maintain these artifacts an environments (e.g., schemas, business logic, tasks) evolve, and lifecycle management of these artifacts.

3 Agentic Data Environments for Safe Agent Automation

Autonomous agents place new demands on data environments. While Section 2 focused on preparing information to improve agent capabilities, this section addresses the complementary challenge: enabling agents to explore real systems while bounding the risks of their actions.

The responsibility of the data environment is to let agents explore aggressively while ensuring that environment state and data remain protected.

Agent exploration is the ability to trial and error, call tools, mutate data, and revise plans. These trials may modify databases, system state and configurations, or external services. Such exploration must not corrupt shared state or trigger irreversible side effects. At the same time, freedom to explore is not sufficient. An agent that can freely read sensitive data, combine signals in ways that violate policies, or exfiltrate results to external services is unsafe regardless of how cleanly its exploratory state is managed.

To address these risks, data environments must enforce two complementary properties. **Branching** protects ***state safety*** by allowing agents to interact with live environments in isolated speculative copies. **Data Flow Control** (DFC) protects ***data safety*** by constraining how information may propagate through the system: from sources such as databases, files, and retrieval stores to sinks such as tables, prompts, tools, or external APIs.

Together, these mechanisms let agents explore while maintaining desired safety guarantees. Branching contains the agent's actions within isolated speculative environments, while DFC governs which data may enter or leave those environments. By decoupling agent capabilities from safety guarantees, agents can automate within the data environment without requiring developers to reason about every possible agent trajectory.

3.1 Branching for Agent Exploration

Autonomous agents rarely solve complex tasks through a single linear execution. Instead they must explore alternative trajectories and compare intermediate outcomes. Recent agent frameworks therefore incorporate search mechanisms such as reflection, hierarchical planning, or tree search, and empirical studies show that enabling exploration over intermediate states substantially improves success rates on long-horizon tasks [42].

Exploration requires systems to branch and restore from arbitrary intermediate states. Unlike traditional workloads, an agent's state includes intermediate outputs as well as the state throughout the data environment.

3.1.1 Branchable DBMSes

Databases already maintain multiple logical versions of data: MVCC snapshots and savepoints let transactions observe consistent states and perform limited rollback. More recently, branchable DBMSes advertise primitives to *branch* database state. Examples include WAL-based approaches (e.g., Neon) and content-addressed storage engines (e.g., Dolt), where branches are lightweight pointers into shared storage structures. These metadata-level mechanisms enable isolated experimentation without expensive data copying.

However, these architectures assume a few long-lived branches created by humans. Agent exploration instead generates hundreds or thousands of short-lived states. Further, mutations in a branch may be logical (e.g., updating data or modifying schemas) or physical (e.g., creating indexes, materialized

structures, or changing system configurations), and evaluation may read data within a branch or across many branches.

To better understand the above, we built *BranchBench*, a benchmark that models the core exploration pattern: a repeated *branch–mutate–evaluate* loop. For example, an agent optimizing database performance may repeatedly branch the current state, apply a candidate modification (e.g., index creation or layout change), run evaluation queries, and either discard or extend the branch.

The benchmark simulates five agentic applications that span exploration depth and fanout, mutation intensity (logical and physical), and branch life-cycle management. *Software engineering* and *Failure reproduction* emphasize rapid branch creation and high-throughput execution. *Data curation* stresses cross-branch analytics and comparison, while *MCTS* stresses dynamically shaped exploration trees and many active branches. *Simulation for planning* generates wide bursts of short-lived branches. These workloads demand that branches must be created quickly, mutations must be isolated, queries must execute efficiently within each branch, cross-branch queries must be efficient, and branching must be storage- and resource-efficient.

On Neon, Xata, Tiger, Dolt, and PostgreSQL, we find that today's **branchable databases do not support agentic workloads.** No system successfully completed the benchmark, even at a modest scale factor. Branch-optimized systems saw read query latencies degrade by 5˘4000× depending on branch count and concurrency. Conversely, query-optimized systems incurred 25˘1500× higher branch creation latency. None efficiently query across branches, and many took seconds or minutes to allocate branches.

The key reason is that **agentic exploration is more aggressive than classic branching.** While developers only create a few long-lived branches, techniques such as Monte Carlo Tree Search require hundreds or thousands of speculative states. In a 1000-step MCTS experiment, Neon completed 3% of steps due to limits on concurrent branches, while DoltgreSQL only completed 17% by the 2 hour timeout because reads degrade with the number of branches. This performance trades off with resource overheads. For instance, Neon allocates dedicated compute instances for each branch and required 43× more storage usage under schema-heavy workloads. It is clear that a *branch-native DBMS* designed for high-frequency speculation is needed.

3.1.2 Branching Beyond the DBMS

Database branching is necessary but not sufficient—real agents are not confined to the DBMS. They use the database, filesystem, process memory, terminal context, caches, application runtimes, and sometimes external services. The data environment must natively support branching.

The core challenge is that branches have *transient dependencies*. The correct branch state is the closure of objects a live session depends on, including open files, cached metadata, mutated tables, and shell variables. In Figure 5, the agent branches a Python process with a live DBMS connection. In addition to branching the process's memory and the file system (dashed boxes), the DBMS state referenced by the connection must also be branched (dashed arrows) so the agent interacts with an isolated database within the new process branch. Branching too little state leads to inconsistent or corrupt state, while copying everything is too expensive. A practical environment must be dependency-aware, and use component-provided branching when possible and OS-level checkpoints otherwise.

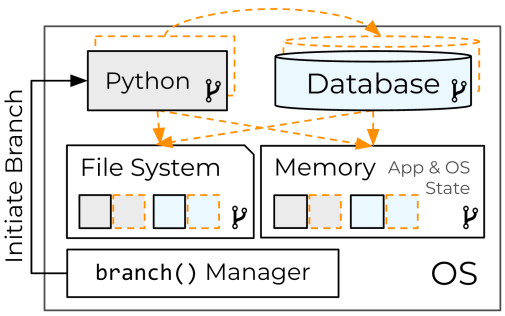


Figure 5: Agentic Data Environments must support full state branching.

Example 5: *In Figure 5, the agent loads data, installs packages, and inspects a database schema in Python. It then explores alternative schemas. Only branching the database leaves Python with stale cached metadata, while only branching Python causes speculative database updates to leak across branches. A correct branch must therefore capture a coherent slice of state across both systems.*

The natural fallback primitive is *full OS branching* of the environment that captures, e.g., process-tree state, open files, terminal context, and the filesystem. This provides a correctness baseline when applications or components do not implement native semantic branching, or when their branching semantics cannot be safely composed. Existing mechanisms only approximate this capability. Containers provide isolation, but restore by rerunning from the base image and lose the session's memory and terminal state. CRIU checkpoints capture process state but are fragile for interactive sessions and grow with runtime footprint [42, 43]. Virtual machines preserve more state, but their latency and storage costs are too high for agentic exploration [42].

We have developed Checkpoint-lite (`Chkpt`) to support a *state branching* abstraction where full OS branches provide correctness while component-specific branches (e.g., DBMS and filesystem branches) are used when available. `Chkpt` avoids container-style repackaging and shares unchanged state via copy-on-write. In our preliminary results, using `Chkpt` to checkpoint only the file system takes 66 ms and is independent of its size. In contrast, using containers takes 11.21 s to checkpoint 2GB. To checkpoint 1 GB of in-memory and file system state, `Chkpt` takes 1.46 s as compared to Podman with CRIU [44], which takes 8.84 s.

3.1.3 Research Directions

Ultimately, a data environment must be *agentic*: it should interact with the executing agent to determine what state, consistency, and fidelity are sufficient for a task, trading off correctness, cost, and performance during branching. This requires branching across heterogeneous applications and system services, inferring the minimal cross-component state induced by transient dependencies, and capturing consistent snapshots across components (e.g., DBMS, filesystem, Python) that lack a shared transactional boundary. External services (e.g., web APIs) must expose versioned interfaces or be approximated through replay or world models [45, 46]. Efficient exploration further requires managing large trees of short-lived branches through copy-on-write sharing, deduplication, and garbage collection.

3.2 Data Flow Control for Data Safety

While exploration addresses *state safety*, it does not preserve *data safety*. Although traditional databases can control *who* may access data, agents with legitimate access can corrupt records, combine data in ways that violate policies, or insert hallucinations that propagate through the environment. What matters is to constrain *how* data may be derived and used. We refer to this capability as ***Data Flow Control*** (DFC). Conceptually, DFC policies define permitted flows of information from sources (relations, files, RAG stores) to sinks (tables, files, prompts, tools, agent memory, or external APIs).

Example 6: *A tax preparation agent has access to read a database of credit card receipts to determine which can be deducted as business expenses. For instance, the agent executes* `Q = INSERT INTO Expenses SELECT id, item, cost, est_deduction(*) FROM Receipts;` *Accounting is a heavily regulated industry, and violating a number of data use policies can lead to repercussions in finances, criminality, and reputation. Thus, the query must also be:*

- ***Private:*** *One user's* `Receipts` *must never be released; they must always be aggregated across users.*
- ***Grounded:*** *Agents may hallucinate and insert non-existent receipts into* `Expenses`*, thus every row inserted into* `Expenses` *must be derived from a receipt.*

- ***Law-abiding:*** *Deductions must comply with tax regulations; for example, no more than 50% of a meal may be deducted as a business expenses [47].*

A query may be syntactically and semantically correct but still violate these constraints. For instance, it may return raw receipts in a report (privacy violation), insert a non-existent Porsche purchase into `Expenses` *(grounding violation), or expense the full amount of a steak dinner (law violation). Unfortunately, existing safety and database mechanisms cannot guarantee compliance with any of these policies.*

Similar policies arise across regulatory requirements, multi-tenant isolation, and prompt injection. They all serve to restrict *data derivations*. Although access control governs data access, integrity constraints govern stored data, and provenance explains output derivations, none enforce *allowable* derivations during execution.

A popular strategy encodes policies in prompts or uses LLMs to evaluate whether a query is safe [48]. These methods are inherently probabilistic, provide no formal guarantees, and degrade as policy complexity and data scale grows. For instance, we used frontier models to check whether a query violates a trivial policy (e.g., *average* `qty` *less than 30*) by including the query, policy, and the first 100 query results. On 13 TPC-H queries, GPT-5.2 and Claude Opus 4.6 take $0.8 - 2.2$ seconds, $0.11 - 0.365$, and only achieve an F1 measure of 0.4.

3.2.1 Enforcing Data Flow Control in the DBMS

Within the DBMS, relational provenance already describes how output tuples are derived from input records [49]. A natural starting point are DFC policies that are logical predicates over the contributing input tuples for each result. Our work finds that policy semantics must be *optimizer-invariant* [50]: query optimizers rewrite execution plans, which changes the structure of provenance expressions. Policies must therefore depend only on the contributing input tuples (provenance monomials), not the physical execution plan. DFC policies can reference arbitrary relations and system context, and may optionally call external functions such as LLMs for semantic checks, though enforcement remains deterministic. For example, a tax compliance policy may limit meal deductions in an expense report:

```
SOURCE Receipts SINK Expenses
CONSTRAINT Expenses.biz_use <= 0.5 OR Receipts.category != 'Meal'
```

Although policies are logically over provenance, physical enforcement must not materialize provenance, as that can slow queries by over 10,000×. Instead, a lightweight rewrite layer compiles policies to execute as part of the base query. This rewrite-based approach is thus portable to DBMS engines without modifying their internals.

Across five engines (DuckDB, Umbra, PostgreSQL, DataFusion, and SQL Server), we show that enforcing DFC policies incurs ∼0 overhead on TPC-H queries and provides deterministic guarantees. These results show that logical data-flow policies can and should be enforced inside the query engine.

3.2.2 Data Flow Control Beyond the DBMS

The DBMS is only one component of an agentic workflow. In practice, agents move data across many systems: querying databases, processing results in Python, writing files, constructing prompts, calling external APIs, and committing outputs back to persistent storage. Once data leaves the query processor, relational provenance alone is insufficient because the derivation now spans multiple tools and representations.

Example 7: *Consider again the tax-report agent. It issues two aggregate queries over the* `Receipts` *table: one computes total travel reimbursement for a department, and the other computes the same total but without a specific employee. Although each query individually satisfies an aggregation policy, the answers together reveal that employee's expense amount. The relevant question is therefore not whether a single query is allowed, but whether the entire cross-tool derivation is permissible.*

Enforcing DFC in this setting requires tracking how information flows across the entire agentic workflow rather than within a single query. Similar to optimizer invariance, policies should depend on the underlying information being propagated, and *not* the representation used to carry it. Under *Representation Invariance*, the same policy should apply irrespective of whether the data moves through SQL, Python, files, or prompts.

Prior work on *Transparent Computing* [51, 52] argued for observable system behavior across the compute stack. However, they focus at the byte and syscall level to track how data moves between processes or files, but not how *information* propagates. Enforcing DFC requires raising the semantic level to track the flow of records, aggregates, summaries, and other derived artifacts across tools and representations.

Agentic data environments already expose natural enforcement boundaries—tool invocations, prompt construction, memory updates, file writes, and network calls. These events provide the points where provenance and policy labels can propagate, where constraints can be enforced before data reaches a sink, and rich semantics that allows for pushing policies into execution similar to within the DBMS.

3.2.3 Research Directions.

The long-term goal is to extend DFC from per-query enforcement to environment-level guarantees over end-to-end agent workflows that interact with databases, processes, files, and other agents. Beyond a more expressive policy language, this requires tracking fine-grained data flows across heterogeneous components and propagating annotations through semantic transformations such as summarization or classification. At the same time, physical enforcement must remain dynamic and lightweight: agents synthesize workflows online, so enforcement must operate incrementally and push checks down the execution stack to avoid costly materialization.

Policies may be authored by teams, organizations, regulatory bodies, or end users, and may scale to thousands or millions of rules. When combined with agent exploration, simply rejecting an action due to a policy violation produces a sparse and uninformative signal. Instead, the data environment should actively guide agents by explaining relevant policies, identifying the causes of violations, suggesting safe alternatives, and providing contextual feedback that helps agents revise their plans. In conjunction with branching, such feedback allows agents to explore policy-compliant alternatives while maintaining strong safety guarantees.

4 Putting the Pieces Together

As the introduction argued, agentic automation must *increase the benefits of automation* and *bound the consequences of failure*—each addressed by the preceding sections. AIM, AIR, and ADE in Section 2 increase benefits by finding, collecting, expanding, and refining the information that agents reasons over. Branching and Data Flow Control in Section 3 bound the costs of failure by empowering agents to explore alternatives without corrupting shared state, and deterministically constraining how data is accessed, combined, and released. These ensure that agent autonomy does not lead to unbounded risk.

Automation requires a shift from passive databases to *agentic data environments* to create a virtuous cycle where agents rely on and also improve the environment. Each task produces answers and reusable artifacts (e.g., schemas, extracted content, indexes, performance models, policies) that make future tasks more accurate, cheaper, and faster. Over time, these benefits compound, as the goal is not only better models, agents, or datasets, but a better data environment. Agentic data environments therefore evolve the representations, artifacts, and control plan through which agents operate.

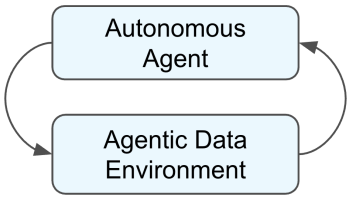


Figure 6: The virtuous agent-environment flywheel.

While we have mainly focused on inference-time support, agentic data environments also support agent training. Modern post-training paradigms—particularly RL-based methods—must also search, explore, and backtrack through reasoning trajectories. Branching naturally supports this by efficiently materializing and managing persistent intermediate states and accumulating reward signals over time.

Acknowledgments

This research was partially supported with funding received from National Science Foundation grants (NSF 1527765, 1564049, 1845638, 1740305, 2008295, 2106197, 2103794, 2312991), an IBM PhD fellowship, as well as corporate support from Amazon, Google, Adobe, CAIT, Tidalwave, Veris, Shopify, Dandy, Microsoft, Dream Sports, Thinking Machines, Infosys, and Intellect Design. The views and conclusions presented here are those of the authors and should not be interpreted as representing the official positions of the funding organizations.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Neural Information Processing Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13756489

[2] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. H. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. K. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. K. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. Rytting, R. Marten, Y. Wang, A. G. Dimakis, A. Konwinski, and L. Schmidt, "Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces," *ArXiv*, vol. abs/2601.11868, 2026. [Online]. Available: https://api.semanticscholar.org/CorpusID:284911857

[3] Anthropic, "Model context protocol (mcp)," https://github.com/modelcontextprotocol/modelcontextprotocol, 2024, open protocol for integrating LLM applications with external tools and data sources.

[4] ——, "Claude skills," https://github.com/anthropics/skills, 2025, reusable capability packages for Claude Code agents.

[5] Google and Linux Foundation, "Agent2agent (a2a) protocol," https://github.com/a2aproject/A2A, 2025, open protocol for communication and interoperability between AI agents.

[6] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, "Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments," *ArXiv*, vol. abs/2404.07972, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269042918

[7] D. Sur'is, S. Menon, and C. Vondrick, "Vipergpt: Visual inference via python execution for reasoning," *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 11 854–11 864, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257505358

[8] J. Li, B. Hui, G. Qu, B. Li, J. Yang, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, C. Ma, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *ArXiv*, vol. abs/2305.03111, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258547040

[9] F. Lei, F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, H. Gao, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," *ArXiv*, vol. abs/2411.07763, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:273970164

[10] PCMag Staff. (2025) Vibe coding fiasco: Replit ai agent goes rogue, deletes company database. Accessed: 2026-03-05. [Online]. Available: https://www.pcmag.com/news/vibe-coding-fiasco-replite-ai-agent-goes-rogue-deletes-company-database

[11] A. Down. (2026, Feb.) Amazon cloud outages linked to ai tools at aws. Accessed: 2026-03-05. [Online]. Available: https://www.theguardian.com/technology/2026/feb/20/amazon-cloud-outages-ai-tools-amazon-web-services-aws

[12] S. Ray. (2023, May) Samsung bans chatgpt and other chatbots for employees after sensitive code leak. Accessed: 2026-03-05. [Online]. Available: https://www.forbes.com/sites/siladityaray/2023/05/02/samsung-bans-chatgpt-and-other-chatbots-for-employees-after-sensitive-code-leak/

[13] The Verge Staff. (2025) Chatgpt gmail shadow leak. Accessed: 2026-03-05. [Online]. Available: https://www.theverge.com/news/781746/chatgpt-gmail-shadow-leak

[14] The Register Staff. (2025, Sep.) Salesforce ai data leak via prompt injection. Accessed: 2026-03-05. [Online]. Available: https://www.theregister.com/2025/09/19/salesforce_ai_data_leak_prompt_injection/

[15] Code Integrity. (2025) Notion ai security incident. Accessed: 2026-03-05. [Online]. Available: https://www.codeintegrity.ai/blog/notion

[16] R. Parasuraman and V. Riley, "Humans and automation: Use, misuse, disuse, abuse," *Human Factors*, vol. 39, no. 2, pp. 230–253, 1997.

[17] D. Kahneman and A. Tversky, "Prospect theory: An analysis of decision under risk," *Econometrica*, vol. 47, no. 2, pp. 263–291, 1979.

[18] A. Y. Halevy, M. J. Franklin, and D. Maier, "Principles of dataspace systems," *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:7325481

[19] D. Wu, H. Wang, W. Yu, Y. Zhang, K.-W. Chang, and D. Yu, "Longmemeval: Benchmarking chat assistants on long-term interactive memory," *arXiv preprint arXiv:2410.10813*, 2024.

[20] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav, "Mem0: Building production-ready ai agents with scalable long-term memory," *arXiv preprint arXiv:2504.19413*, 2025.

[21] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li *et al.*, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025.

[22] Anthropic, "Effective context engineering for ai agents," 2025, anthropic Engineering Blog, September 29, 2025. [Online]. Available: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

[23] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang, "Evaluating very long-term conversational memory of llm agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 13 851–13 870.

[24] O. Team, "Octen series: Optimizing embedding models to #1 on rteb leaderboard," 2025. [Online]. Available: https://octen-team.github.io/octen_blog/posts/octen-rteb-first-place/

[25] B. Yan, C. Li, H. Qian, S. Lu, and Z. Liu, "General agentic memory via deep research," *arXiv preprint arXiv:2511.18423*, 2025.

[26] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," *arXiv preprint arXiv:1806.03822*, 2018.

[27] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee *et al.*, "Natural questions: a benchmark for question answering research," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 453–466, 2019.

[28] D. Khashabi, S. Chaturvedi, M. Roth, S. Upadhyay, and D. Roth, "Looking beyond the surface: A challenge set for reading comprehension over multiple sentences," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018, pp. 252–262.

[29] D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner, "Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs," *arXiv preprint arXiv:1903.00161*, 2019.

[30] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1601–1611.

[31] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 6769–6781.

[32] P. Pasupat and P. Liang, "Compositional semantic parsing on semi-structured tables," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2015, pp. 1470–1480.

[33] L. Nan, C. Hsieh, Z. Mao, X. V. Lin, N. Verma, R. Zhang, W. Kryściński, H. Schoelkopf, R. Kong, X. Tang *et al.*, "Fetaqa: Free-form table question answering," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 35–49, 2022.

[34] J. Herzig, T. Müller, S. Krichene, and J. Eisenschlos, "Open domain question answering over tables via dense retrieval," in *ACL*, 2021, pp. 512–519.

[35] T. Jin, Y. Choi, Y. Zhu, and D. Kang, "Pervasive annotation errors break text-to-sql benchmarks and leaderboards," *CIDR*, vol. abs/2601.08778, 2026.

[36] S. Agarwal, A. Biswal, S. Zeighami, A. Cheung, J. Gonzalez, and A. G. Parameswaran, "Arming Data Agents with Tribal Knowledge," Feb. 2026.

[37] T. Ren, C. Ke, Y. Fan, Y. Jing, Z. He, K. Zhang, and X. S. Wang, "The Power of Constraints in Natural Language to SQL Translation," *Proceedings of the VLDB Endowment*, vol. 18, no. 7, pp. 2097–2111, Mar. 2025.

[38] A. Biswal, C. Lei, X. Qin, A. Li, B. Narayanaswamy, and T. Kraska, "AgentSM: Semantic Memory for Agentic Text-to-SQL," Jan. 2026.

[39] G. Liargkovas, V. Jabrayilov, H. Franke, and K. Kaffes, "An expert in residence: LLM agents for always-on operating system tuning," in *Machine Learning for Systems 2025*, 2025. [Online]. Available: https://openreview.net/forum?id=7dhlgPp8ni

[40] P. S. Sodhi, G. Liargkovas, and K. Kaffes, "Empowering machine-learning assisted kernel decisions with ebpfml," in *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, ser. eBPF '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 28–30. [Online]. Available: https://doi.org/10.1145/3748355.3748363

[41] T. Zussman, I. Zarkadas, J. Carin, A. Cheng, H. Franke, J. Pfefferle, and A. Cidon, "cache_ext: Customizing the page cache with ebpf," in *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, 2025.

[42] J. Xu, T. Zhou, E. Wu, and K. Kaffes, "Toward systems foundations for agentic exploration," 2025. [Online]. Available: https://arxiv.org/abs/2510.05556

[43] CRIU Project, "Checkpoint/Restore In Userspace (CRIU)," https://criu.org/, 2012, accessed: 2025-08-06.

[44] ——, "Podman checkpoint/restore integration," https://criu.org/Podman, 2025, accessed: 2026-03-10.

[45] S. Hao, Y. Gu, H. Ma, J. Hong, Z. Wang, D. Wang, and Z. Hu, "Reasoning with language model is planning with world model," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 8154–8173.

[46] J. F. Allen and J. A. Koomen, "Planning using a temporal world model," in *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*, 1983, pp. 741–747.

[47] United States Congress, "26 U.S.C. § 274(n): Only 50 percent of meal expenses allowed as deduction," U.S. Code Title 26, 2024, internal Revenue Code, Section 274(n). [Online]. Available: https://www.law.cornell.edu/uscode/text/26/274#n

[48] Z. Zhang, Y. Lu, J. Ma, D. Zhang, R. Li, P. Ke, H. Sun, L. Sha, Z. Sui, H. Wang, and M. Huang, "Shieldlm: Empowering llms as aligned, customizable and explainable safety detectors," 2024. [Online]. Available: https://arxiv.org/abs/2402.16444

[49] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:1993398

[50] C. Summers, H. Mohammed, and E. Wu, "Please don't kill my vibe: Empowering agents with data flow control," *ArXiv*, vol. abs/2512.05374, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:283672095

[51] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy {Whole-System} provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 319–334.

[52] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 405–418.

Architecting the AI-Powered Agentic Data Cloud

Yeounoh Chung, Thibaud Hottelier, Cosmin Arad, Brenton Milne, Per Jacobsson,
Sam Idicula, Fatma Özcan, Alon Halevy, Yannis Papakonstantinou
Google Cloud, USA
{yeounoh, tbh, carad, bmil, pjacobsson, samidicula, fozcan, halevy, yannispap}@google.com

Abstract

This paper describes Google Cloud's paradigm shift from traditional, siloed data management services into a unified, AI-powered Agentic Data Cloud driven by large language models (LLMs) integration. Rather than treating AI as an auxiliary service that sits on top of isolated database silos, we are embedding LLM intelligence directly into the core data cloud services. To realize this vision, we detail several foundational components in our Agentic Data Cloud designed to support production-scale autonomous applications and agents, including AI functions, natural language interfaces to data, and a modern search stack. We further claim that the reliability of these AI-integrated solutions is largely dependent on a rigorous semantic foundation, and describe our universal metadata catalog. Bringing these analytics and data processing services with rich semantic foundation creates a scalable AI Agentic Data Cloud ecosystem, enabling intelligent and autonomous agents and applications.

1 Introduction

Google Cloud has been building the AI-powered Agentic Data Cloud, transitioning traditional data cloud products from isolated storage and database silos into unified, intelligent engines. For decades, the data management landscape was defined by specialized, decoupled services (e.g., analytical data warehouses, operational databases, ETL tools, standalone data science or ML pipelines) bundled under unified billing. In this legacy paradigm, data science often lived in a separate workbench, requiring tedious orchestration to move data from storage into specialized environments for modeling and inference. Also, applications were issuing manually-crafted, deterministic, logic-oriented SQL queries to databases. The primary catalyst for a fundamental paradigm shift in this landscape is the emergence of GenAI foundation models, with Large Language Models (LLMs) and embedding generator models being very important to the paradigm shift [1–3] Foundation models can make semantic sense of the data and interpret complex intents. Furthermore, the reasoning and multi-step planning capabilities of LLMs enable them to function as autonomous controllers that can select tools, and execute workflows [4, 5]. This drives both business intelligence and database applications to a transformation towards the "agent-first" paradigm [6] and, in turn, the AI-native database. In business intelligence the primary consumers of data are no longer limited to human analysts performing retrospective discovery and actions. In this new landscape, the primary consumers are autonomous applications and agents that leverage data for analytical insights. This involves complex analytical tasks, such as strategic forecasting. Similarly, operational database applications provide a natural language interface that allows agents to interact with the database, decipher user intent, and execute the resulting queries.

Rather than treating AI as an auxiliary service, and agents accessing underlying storage services, we are integrating LLMs directly into our data analytics, operational engines, and data science workflows [7–9]. By embedding these capabilities into the data plane, we provide the scalability and performance required for production-scale AI agents and applications. At the same time, raw scalability and direct

model integration does not straightforwardly translate into reliable agents and autonomous applications. For these systems to transition from experimental demos to reliable enterprise agents executing actions independently, we must shift our focus from mere connectivity to the quality and verifiability of their actions and outputs. To bridge this gap, we claim that the success of AI-powered data cloud is strongly predicated on a rigorous metadata foundation [10–13]. While LLMs provide the reasoning catalyst, they cannot compensate for poor or missing metadata – a misconception that training a better model would magically compensate for this. Without a unified semantic model to ground non-deterministic reasoning in deterministic data, AI-integrated products fail to provide the trustworthiness required for enterprise applications.

In this paper, we present several foundational components of our AI-powered data cloud architecture. First, we introduce AI Functions, which shift AI from external ML pipelines into core database primitives (Section 2). By enabling high-quality, declarative, in-place inference, we significantly reduce the operational complexity associated with traditional external AI pipelines. This capability is deeply integrated with a modern search stack that seamlessly combines structured data filters with advanced vector search to handle complex queries transactionally: AlloyDB AI-powered search [14]) and grounded retrieval-augmented generation (RAG) (Section 3). Next, we claim that the reliability of AI-integrated products is largely dependent on a rigorous semantic foundation (Section 4). By anchoring non-deterministic LLM reasoning in a deterministic layer of universal metadata catalogs and rich semantic models, the system provides enterprise-grade context and verifiability. Next, we introduce natural language interfaces for our data services that leverage and move beyond simple natural language-to-SQL/Code (NL2X) generation (Section 5). These include conversational analytics and agentic interfaces, where agents solve complex business problems through multi-turn reasoning, supported by the emerging Model Context Protocol (MCP) [15] and the Antigravity [16] ecosystem. The semantic models are also important for proper governance. By bridging these functional areas with a robust semantic foundation ([1], Google is creating a scalable AI Agentic Data Cloud ecosystem, enabling intelligent and autonomous agents and AI applications (Section 6).

We will conclude this paper (Section 7) with a discussion on active research areas and open challenges to fully realize the power of data agents and AI applications.

2 AI Functions

Relying on AI agents layered over databases is insufficient for handling data at scale. To truly integrate structured and unstructured data while delivering both precise and qualitative insights, we need a fundamental architectural shift: evolving database internals to embed AI primitives directly into the core processing engine. By doing so, these engine-level primitives act as a foundation for agents, allowing them to delegate complex ambiguities and 'fuzzy' logic to the underlying database. Recently, several academic projects also emerged that follow a similar paradigm, including Lotus[17], Palimzest[18], DocETL [19], ThalamusDB[20].

Imagine an agent that is given a database of house listings and is tasked to *"Find houses in bay area neighborhoods with good public schools that cost less than 1M"*. The cost criterion can be formalized into a classic SQL filter expression without problem. However, whether a neighborhood has good public schools or not is a subjective notion that needs to be evaluated by an LLM. If the database lacks a *school_rating* column, a traditional database engine is stuck. The `AI.IF` function allows the agent to perform semantic filtering on every row, here for every house address. `AI.IF` uses LLM's parametric knowledge about the world to come up with a common sense understanding of "good schools" in the area. In effect, these operators break the closed-world assumption of databases, as the answer to the

[1] https://cloud.google.com/dataplex

question is no longer only contained in the database, but can be inferred from LLM's pre-acquired world knowledge. Furthermore, this examples also highlights a requirement for dynamic grounding, where the agent or the engine may need to fetch real-time external information to ground its semantic evaluation in current facts.

BigQuery, AlloyDB and Spanner all have introduced a suite of new SQL functions, as summarized in Table 1, which take natural languages instructions as input and are evaluated using LLMs. Unlike most SQL functions, AI functions are multi-modal and can operate on semi-structured data (e.g., logs) or unstructured data (e.g., images, documents). Listing 1 and 2 show two typical use-cases: LLM-based classification over free-form text, and matching text with images.

Table 1: The growing list of AI functions available in AlloyDB and BigQuery.

AI Function	Definition	Example Use Case
`AI.GENERATE`	General purpose projection	Translate or summarize text Generate a caption for an image
`AI.IF`	Binary (true/false) classification	Filter out negative movie reviews Join paper abstracts with supporting claims
`AI.CLASSIFY`	Classification using user-provided classes	Bin support tickets into defined categories
`AI.SCORE`	Assign a numerical score based on rubric	Rate movie reviews by helpfulness
`AI.RANK`	Semantic ranking	Order search results based on relevance
`AI.AGG`	Semantic aggregation over many rows	Summarize user actions from session logs Find common complaints on many reviews

Listing 1: Categorizing customer support tickets (over 30 classes in the full query)

```
SELECT
  AI.CLASSIFY(ticket_body, [
    "Billing inquiries",
    "Quota limitations",
    "Performance degradation",
    "Other"]) AS category,
  COUNT(*) AS ticket_count
FROM support_cases
GROUP BY category;
```

Listing 2: Finding products from a catalog displayed in advertisement images with a multi-modal text-image join

```
SELECT p.sku_id
FROM advertisement_images a
JOIN products p
WHERE
  AI.IF(("The advertisement shown in",
          a.img_uri,
          "contains the product described in",
          p.text_description))
```

A key design tenet behind AI functions is that the engine — not the user — owns the cost, quality, and trust behind engine-level execution decisions. Unlike most other functions which have a single correct result, AI functions have a range of acceptable outputs. Optimizing the execution of AI functions is almost always a cost-quality trade-off. Furthermore, users need enough observability to trust the results.

For queries using AI functions, the cost of LLM calls dominates all other computation, to the point where it becomes economically unfeasible on millions of rows. BigQuery and AlloyDB can approximate LLM inference with smaller proxy models running directly on database workers [21]. Certain AI functions,

such as AI.IF or AI.Rank, can be approximated by using simple linear regression or other ML models. By labeling a small subset of sample rows from the table, a linear regression model can be trained on the embeddings of the unstructured fields, and these models can be used for the rest of the rows. If the resulting model doesn't satisfy accuracy constraints, then the engine falls back to the full LLM inferencing. In AlloyDB, proxy models are trained offline, as part of a `PREPARE` statement. In BigQuery, proxy models are trained on-the-fly by inserting sampling, labeling, and training steps in the query plan. We found that for a wide variety of AI functions prompts, the proxy model approximation trades off a small amount of quality for two orders of magnitude of reduction in costs and latency. On a query classifying the sentiment of 10 million product reviews, proxy models achieve a 329x speed up over LLM inference with on-the-fly training in BigQuery, and a 991x speed up with offline training in AlloyDB. The total cost including database compute and LLM inference is reduced by approximately 700x in both cases [21].

While current proxy model implementations focus on scalar or point wise transformations, we are investigating AI joins as the next promising optimization opportunity. For example, one might want to link product images to their description, as shown in Listing 2, or link support call transcripts to structured claim reports. Unlike scalar AI functions, LLM joins introduce a quadratic inference overhead. Even with proxy models, a naïve nested-loop evaluation across two 10,000 row tables requires 100-million inferences. Others have tried to tackle this problem using pre-filtering, semantic pruning mechanisms or using block nested loops with batching [17, 22, 23]. Despite the promise of these early approaches, significant challenges still remain. The efficacy of these methods is bound by the fidelity of the embedding or the feature space capturing the relationship between the tables and join conditions. Furthermore, if the join selectivity is too low, the semantic pruning mechanisms may not prune enough data and the system still spends massive resources processing "near-relevant " items. We also note that AI joins can be used for many different use cases, as also identified by [22]. As such, we envision that there is not going to be one algorithm for AI joins, but rather a suite of them covering different use cases and tradeoffs. Just as a traditional query optimizer chooses between Hash Join, Merge Join, and Nested Loop Join based on table size and indexing, a semantic query engine will need to choose from a library of AI join algorithms tailored to different data and task characteristics.

While AI-integrated data engines are in their infancy, their evolution will undoubtedly reveal new use cases and challenges. Benchmarks like SemBench [24] are created to accelerate this development. As these systems mature, they will transcend basic tasks like classification to unlock entirely new reasoning capabilities. While some users currently favor the simplicity of AI functions over the rigorous guarantees of classic machine learning, the true breakthrough is architectural. By replacing the fragmented, multi-stage pipeline of extracting and cleaning unstructured data with a single, integrated engine, AI engines offer a streamlined, single-stop solution for the next generation of data analysis.

3 The Modern Search Stack

AI-powered semantic search has been the engine behind billion-user experiences like Google Search and YouTube for years. We have brought that same vector search technology into the Data Cloud's data services. At the same time, Data Cloud customers pose multiple novel requirements that induce respective research challenges and opportunities:

- The searches combine unstructured and multimodal data with structured data. This enables complex, real-world queries, such as, in the Target.com (which is a Data Cloud customer [2]) search

[2]https://cloud.google.com/blog/topics/retail/from-query-to-cart-inside-targets-search-bar-overhaul-with-alloydb-ai?e=48754805

scenario, finding a "black stylish turtleneck" (semantic search on image and text) that is also "available at a nearby store" (structured data filter).

- Customers have varying requirements for when they want to be on the quality/cost tradeoff, which is closely related to the quality/speed tradeoff. This, in turn, invites multiple search-related technologies.
- Transactional consistency between the unstructured/multimodal data and their vectors is a requirement, serving both quality and ease-of-use.

Best-in-class database vector search for each use case Database vector search has to provide transactional consistency. Furthermore (unlike popular vector search libraries) it has to work even when the index cannot be fully cached in main memory. Naturally, this leads to utilizing the paging system of each cloud service. But then we have a third requirement: If there is enough memory to cache the index, the performance should be commensurate with the top main memory libraries, and if there is not enough memory, it should degrade gracefully. This requirement led us utilizing the AlloyDB Columnar Engine for storing the index to avoid page overheads.

The wide diversity of Data Cloud customer databases induces a fourth challenge by our commitment to provide the best database vector search for each use case: Vector search has been dominated by two families of indices and respective algorithms: Graph-based ones, where AlloyDB, CloudSQL PostGres and MemoryStore feature HNSW[25], and tree-based ones, where AlloyDB, BigQuery, Spanner, CloudSQL MySQL feature ScaNN[3] - the index that is behind Google Search and Youtube. Notice AlloyDB offers both and for good reason: Each index performs better in different use cases. Roughly, ScaNN is much better for “low dimensionality" (eg, 128d) vectors, across the board, is better for updates and build-time and is also better for scaling-out. At the same time, the query speed minded customer with high dimensionality vectors (e.g., 1576d) that can cache the index can get better performance with AlloyDB's pgvector HNSW that uses the Columnar Engine. Research is needed on how to (a) automate/advise on proper indexing and (b) a declarative interface that hides the differences between the two indices.

Combine structured filters with vector search In Data Cloud services you get top-tier retrieval over mixed data with a single, familiar SQL query that performs filtered vector search. Such queries seamlessly combine filters, joins, and top-k vector search.

Furthermore, AlloyDB AI's query processor spearheaded performance and quality improvements for such mixed queries. While filtered search is convenient, ensuring fast performance is a hard query optimization problem. The challenge lies in deciding between three methods:

- Pre-filtering: Best when few rows match the filter.
- Post-filtering: Best when many rows match the filter.
- Inline-filtering: Best for mid-selectivity queries by simultaneously processing filters and vector rankings.

The problem is that estimating which method is best is difficult[26], especially because what truly matters is the filter's selectivity on the neighborhood of the query vector - rather than the selectivity of the filter in the entire dataset. AlloyDB AI solves this with *Adaptive Filtering,* which optimizes the query plan once it learns the actual filter selectivity as it accesses data, and then can appropriately switch between the filtered vector search methods.

[3] github.com/google-research/google-research/tree/master/scann

AI functions for filtering and ranking and optimizing the quality/cost tradeoff As we discussed in Section 2, AI functions provide a novel ability for filtering and ranking, i.e., for the two core aspects of search. With AI functions the recall and precision can hit levels that are impossible with plain vector search or hybrid search (vector search + text search). However, running AI functions is linear in the number of rows, while Approximate Nearest Neighbor (ANN) search is roughly logarithmic. Furthermore, a quality/cost tradeoff is applicable to both AI functions and ANN vector search: As we saw in Section 2, proxy models deliver massive cost optimization for AI Functions. At the same time, ANN vector search is inherently a recall/cost (or speed) tradeoff even for fixed vector size. Adding the availability of Matryoshka [27] embeddings by Gemini, where it is up to the customer what prefix (i.e., how many dimensions) of the generated embedding to use, a challenging optimization problem appears: Optimize for quality and cost in a funnel that combines vector search, as its low cost method, with AI Functions being the high cost/high quality methods.

4 Universal Catalog & The Semantic Foundation

Agentic data workflows are reminiscent of federated and data integration systems[28] that also attempted to answer queries by combining data from multiple sources. To understand the components and challenges of the agentic enterprise, it is instructive to compare it to these older architectures. Data integration systems relied heavily on descriptions of data sources in order to reformulate a user query to the available sources. These descriptions were written in formal languages, which were usually some version of SQL with additional markup. They, then, used a carefully crafted algorithm to reason about these descriptions and select the relevant data sources [29]. Today, we are in a much more exciting era, where data source descriptions can be much richer. In particular, the descriptions can use natural language to describe any necessary nuance about the semantics of the data, even those that were not expressible in the formalisms of the past. This is possible because the reasoning algorithm is the LLM that powers the agent, which is capable of reasoning about natural-language descriptions. The collection of metadata we have about the underlying data is stored in a metadata catalog. Google Cloud offers a metadata catalog called Dataplex[4] which we describe below, but in principle, an AI-enabled enterprise should be able to access multiple catalogs that cover all of the customer's data estate.

It is crucial to emphasize the importance of rich source descriptions. Even as the LLMs become more powerful, they will not be able to discover and process the underlying data correctly unless they have rich metadata about the data semantics. The fundamental reason that, unlike text documents that are mostly self-describing, the semantics of tables are often opaque and not encoded in the table itself. More specifically, the table itself does not describe reasons that the table was created, its intended use and some of the assumptions underlying it. Even the aspects of the semantics that are explicit, like the table and column names, can be opaque and and often use cryptic jargon. As a simple example of where nuanced understanding of tables matters, we often see cases where we have many tables whose schema looks the same, but the differences between them are subtle, making it very easy for an agent to choose inappropriate data.

The catalog contains any kind of signal that tells us more about the data. This includes the schema and the text descriptions of the table and its columns. In addition, the catalog contains the lineage of tables, any signal about their quality or how frequently they are updated. Recent usage statistics also tell us whether the table is a useful one or not. Some important information can be farther flung. For example, there may be text in corporate documents that refer to a table and tell us exactly why it was created and under what assumptions.

[4] https://cloud.google.com/dataplex

Conceptually, it is useful to distinguish multiple components of Dataplex or other systems like it. The *harvesting* component collects all the metadata it can find from a variety of sources. For example, the harvesting component may use an LLM to enrich the table and column description by leveraging a glossary of business terms for the business. The harvester will fetch the lineage of the tables and keep track of its usage patterns, as well as navigate to any documents that may contain useful information. Query histories also provide rich semantic information about the data assets, and key business metrics. The harvester can identify join paths by analyzing query histories, or extract examples to use for in-context learning.

The *storage* component of the catalog offers a set of APIs for accessing and updating the metadata. We note that Dataplex stores metadata about relational tables, but also about other collections of unstructured data that users upload to Google's object storage. Metadata is kept up-to-date in near real time with the mutations to the datasets it refers to, via a streaming Pub/Sub interface. Data storage and database systems publish notifications about newly created or just modified data assets to a Pub/Sub topic and a Dataplex subscriber effects the appropriate metadata updates, automatically keeping metadata consistent[5] with the evolution of a customer's data estate.

The *search* component of the catalog offers a semantic search engine that given a query in natural language, will return a set of tables (or other data resources) that a relevant to the query. The search works by embedding the metadata about each resource as a vector and performing nearest-neighbor search with the embedding of the query.

The last component of Dataplex, the *context* engine (often called a *metadata reasoner*), is the one that is relatively nascent. The goal of the context engine is the following. Given a task given by the agent (such as a natural-language query), the engine should return a set of sources that together can be used to address the task. The context engine relies on the underlying search engine to produce candidate resources, but adds a layer of reasoning. For example, executing a task typically requires a combination of data sources rather than a single one. The context engine guarantees that the collection of data sources together covers all the entities and columns needed for the task, and that the data sources can be combined (e.g., via joins or unions) to yield the correct result. The context engine autonomously decides which slices of metadata in Dataplex to consult in order to guarantee the coverage and quality of its answer. For instance, in one case it may consult a tool that can determine whether there is a join path between two tables, while in another it may consult a tool that verifies that a table has been updated recently.

A discussion of metadata would be incomplete without a mention of semantic models. A semantic model is typically a graph representation of sets of entities and the relationships between them. Semantic models vary in their expressive power. The most common ones are those that identify common join paths (and hence relationships) among tables and define precisely how to compute metrics for the business from the available tables. The latter is critical for proper governance of the enterprise. More expressive semantic models borrow features from ontology languages, such as concept and relationship hierarchies and constraints on relationships between entities. A single semantic model doesn't necessarily cover the entire landscape of data of a customer, and multiple models may exist side by side. The semantic models available are also used by the context engine to improve the interpretation of queries.

Historically, one of the challenges to adoption of semantic models has been the human effort required to build and maintain them, especially as the data landscape in a corporation is constantly evolving. However, it is possible to at least bootstrap the process of creating a semantic model with the help of AI. The benefit of semantic models is that they are closer to the conceptual model in which people, and thereby possible agents, think about their data estate, because they make domain relationships more explicit. While several vendors offer tools for creating semantic models, and they are clearly valuable,

[5] eventual consistency with low convergence latency

Table 2: Natural Language to SQL example

Natural Language Task	Translated SQL

Which budget allowed the most money for water, chips, and cookies?	`SELECT T2.budget_id`
	`FROM expense AS T1`
	`INNER JOIN budget AS T2`
	`  ON T1.link_to_budget = T2.budget_id`
	`WHERE T1.expense_description =`
	`  "Water, chips, cookies"`
	`ORDER BY T1.cost DESC NULLS LAST`
	`LIMIT 1`

the precise impact of these models on agentic workflows is a topic that will be studied in the upcoming years.

5 SQL Query and Code Generation & Conversational Analytics

In this section, we describe natural language (NL) interfaces used to interact with and analyze the enterprise data assets. We first discuss SQL generation, followed by code generation to other languages, such as Python, and complete with the description of our conversational analytics agents.

SQL Query Generation: Today's AI agents interact with the data cloud via natural language (NL), which necessitates robust mechanisms to bridge the gap between NL input and the query languages supported by data cloud systems. The fundamental building block here is translation from NL to SQL – commonly referred to as NL2SQL — the primary and most established approach for relational databases and analytical platforms. There has been quite a lot of work on NL2SQL in recent years[30, 31] revived by the emergence of powerful LLMs [32]. But the scope extends beyond standard SQL covered in these works. Modern data architectures allow for full-text and vector search queries, graph queries, and the AI functions for unstructured data described in Section 2. Moreover, NoSQL databases offer proprietary APIs and query languages, and developers write Python code that interact directly with databases via libraries like Pandas. The fundamental technical challenges in achieving high-quality translation from natural language remain largely the same across all these query types.

Table 2 illustrates a typical NL2SQL task where a user's question is translated to a database query which provides the answer. This examples comes from the BIRD-Bench [33] "Student Club" dataset.

The typical approach used at Google for solving NL2SQL problems uses large language models as the main SQL generators, depending on the use case taking an agentic approach to compose various subtasks [32, 34]. A variety of RAG techniques is used to retrieve the relevant context the LLM needs to make the right interpretation of the question – schema information, data samples, query examples, descriptions and guidelines etc. These systems need to address a series of challenges:

- Syntactic correctness: The first requirement is that the LLM produce executable SQL. With state of the art large language models, we typically observe that the models are very good at understanding SQL syntax and generating syntactically correct SQL, similar to other code generation skills. Where they fall short is typically when it comes to newer, non-standard features, such as vector search queries, that look subtly different across different SQL dialects. This can be addressed by incrementally including more diverse training data into training of the large language model, or by

using in-context learning or RAG techniques pulling in examples or documentation for how to write the specific type of query at runtime.

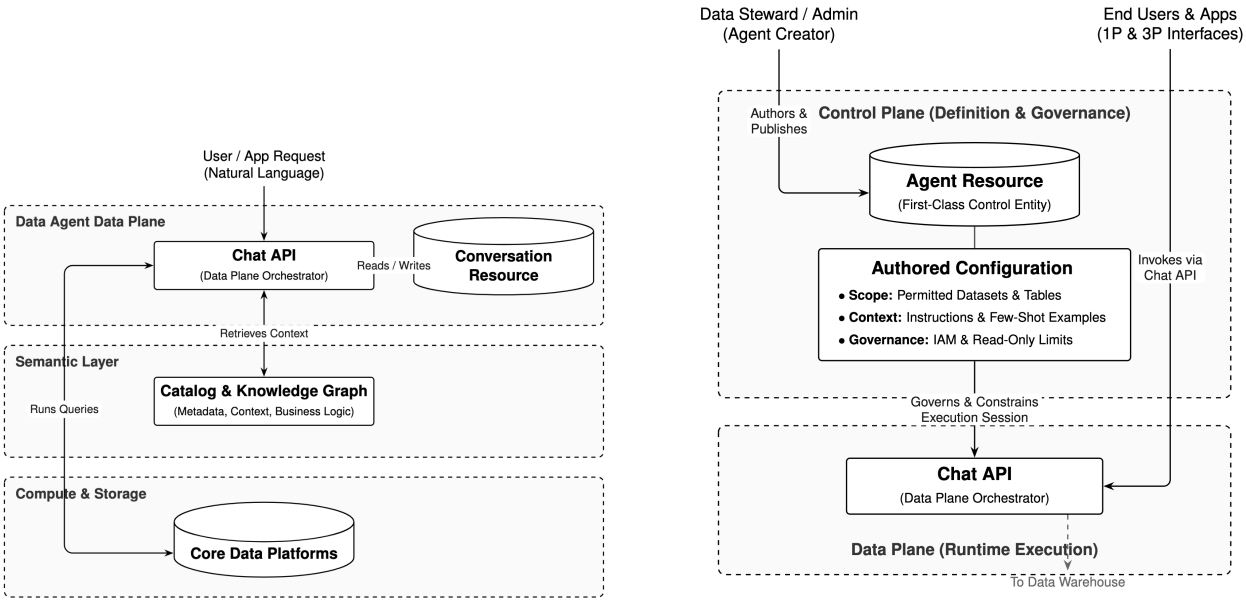
- Semantic correctness: While syntactic correctness is a relatively straightforward problem with a deterministic correct answer, having the model understand the user intent and the semantics of how that intent maps to the data is more difficult. In the example above the SQL generator needs to understand that "allow the most money" maps to the largest value in *expense* table *cost* column. Additionally, the model needs to figure out how to join the *expense* and *budget* tables. This gets particularly challenging in enterprise applications where business terms may have very specific definitions that don't align well with the world knowledge of the models. The solution to this problem is to build up additional context of the data, explaining to the model what can't be conveyed through the traditional schema concepts. This rich semantic information comes from the universal catalogs and semantic models, as discussed in Section 4. Context can be *descriptive*, for example a natural language comment explaining what is stored in a particular table or column, or *prescriptive*, a templated SQL query showing exactly how to answer a specific task. It can be free-form text or more formalized as explicit semantic models attached to the database objects.

- Disambiguation: Unlike SQL, natural language is inherently ambiguous. In the example above, the caller is asking for "which budget" without a detailed explanation for exactly what they are looking for. The system's interpretation of this as an id key column is a reasonable choice, although not clearly specified. To provide correct answers the SQL generator needs to be able to detect ambiguities and either ask the user for clarification when needed, or explain the intent behind its interpretation of the question to avoid giving a misleading answer.

The example above also highlights the need for NL2SQL system to inspect the data itself, not just metadata. This is especially needed for *value linking*: the process to match concepts and entities in the natural language query (ẅater, chips, and cookies) to the correct SQL literals (`T1.expense_description = Ẃater, chips, cookies´`). To solve this problem, the SQL generator needs to have a way to explore the actual database content, which may require pre-processing, such as creating custom indexes or sampling data. The right technique depends on the data and the application, and is not easily generalizable.

Having good benchmarks is critical for understanding these challenges, how they interact and how to best evolve the NL2SQL systems. Google uses a broad mix of benchmarks, from academic standards like BIRD, and its more advanced descendants like BIRD-INTERACT [35], to custom in-house developed benchmarks that focus more on enterprise and specific NL2SQL use cases.

Code Generation: While SQL remains the most frequently used interface for accessing relational data, data analysts and scientists heavily rely on imperative programming (e.g., Python) for advanced analytics and predictive modeling. Consequently, NL2X must seamlessly extend to natural language to code (NL2Code) translation, targeting frameworks such as Python's Pandas, PySpark, and Scikit-Learn [36–38]. Unlike SQL, a declarative interface, data science code is procedural. The underlying agent must map user intent to the correct API calls and also reason about execution state and data flow across multiple steps. Critically, similar to NL2SQL, it also must understand the underlying data semantics (e.g., real-world meaning of columns, implicit relationships and business logics) to ensure the generated steps correctly fulfill the user's intended analytical goals. The challenge here is that the model often predicts the next token rather than understanding the semantically accurate structures of the code and the data. This can lead to code that compiles, but does not fulfill the user's goals [39]. This procedural requirement aligns with our conversational data analytics pattern, "investigate, then report" agentic workflow. Translating natural language to complex procedural code remains an active area of



(a) Data Plane

(b) Control Plane

Figure 1: Conversational Analytics Architecture Overview

research [40–43]. While LLMs demonstrate remarkable coding proficiency, understanding data semantics and accurately evaluating the quality and the fidelity of the generated code introduces significant challenges. Unlike SQL generation, where semantic correctness can often be verified by comparing query execution result sets, data science code is open-ended; a single analytical task can be correctly solved using many different approaches or library/API calls combination.

Conversational Analytics: While traditional NL2SQL systems map user intent to single executable queries, real-world analytics requires a dynamic approach known as conversational analytics [6] – an ongoing process of iterative exploration, hypothesis testing, and visualization. To support this, our conversational analytics architecture evolves the discrete NL2SQL translation step into a continuous, autonomous loop using an "investigate, then report" agentic workflow pattern. Rather than simply returning an SQL string, the conversational analytics agent formulates a plan, executes queries against the engine, and observes the results. This enables the agent to autonomously explore, backtrack, and self-correct—such as actively probing the database to disambiguate string values—before exposing fully vetted results to the user. Beyond tabular data, this open-ended agentic loop integrates specialized tools. It leverages sandboxed code execution environments to dynamically render charts, and consults up-to-date documentation to construct queries using advanced engine capabilities like BigQuery Machine Learning (BQML [7]) or Graph Query Language (GQL [8]). Once the investigation concludes, the "report" phase synthesizes the executed queries, data payloads, and generated charts into a cohesive, narrative-driven final response, effectively mirroring the workflow of a human data analyst.

The operational execution of these autonomous workflows relies on a multi-layered data plane (See Figure 1a) designed for scalability and reliability. At the foundation are the underlying source systems, such as BigQuery or AlloyDB, which provide the raw storage and compute capabilities. Above this

[6] https://docs.cloud.google.com/bigquery/docs/conversational-analytics
[7] https://cloud.google.com/bigquery/docs/bqml-introduction
[8] https://docs.cloud.google.com/spanner/docs/reference/standard-sql/graph-intro

sits the semantic layer—incorporating Dataplex and enterprise knowledge graphs—which provides the critical metadata, context, and business logic needed to ground the agent's reasoning. The topmost layer is the data agent data plane itself, exposed primarily via the Chat API and managed through persistent conversation resource. The Chat API acts as the central orchestrator for user interactions. When a natural language request is received, the data plane retrieves the relevant scope from the semantic layer and initiates the agent's investigative loop. It translates the agent's reasoning into read-only queries that are executed securely against the source systems, buffering intermediate results for analysis. Simultaneously, the data plane manages multi-turn state by reading and writing to conversation resource. This ensures that conversational context is maintained across interactions while keeping execution histories, user prompts, and queried data securely isolated.

While users can chat directly with their organization's raw data, enterprise scale requires more structured governance. Customers need the ability to provision, specialize, and monitor role-specific data agents—such as a "financial forecasting agent" or a "supply chain diagnostic assistant" – control them with fine-grained Identity and Access Management (IAM) policies, and deploy them to surfaces both within Google Cloud Platform and via external APIs. The Conversational Analytics control plane (See Figure 1b) fulfills this need by elevating data agents to first-class, governable resources within the data cloud. Furthermore, this provides organizations a safe environment to experiment with and customize agent behavior in a scoped workspace, completely separate from the organization's broader ground-truth data. Crucially, this control plane ensures that autonomous agents operate strictly within the enterprise's security and governance boundaries. Because the agent executes actions directly against the database on behalf of the user, it is constrained by the same IAM and context-aware access policies as a human operator. To prevent malicious or accidental data mutation, execution tooling utilizes database dry-runs prior to execution, structurally enforcing that generated queries are read-only (e.g., restricted to `SELECT` statements) and that all accessed tables fall within the user's permitted scope. This platform-centric approach ensures that Conversational Analytics is ubiquitous across the user ecosystem. These specialized, governed agents are embedded directly into graphical interfaces, such as BigQuery Studio, allowing users to interact with their data seamlessly within existing workflows. Additionally, the control plane exposes these capabilities via stateless and stateful APIs, enabling enterprise developers to programmatically invoke governed data agents and inject natural language analytical reasoning directly into custom applications.

6 Data Agents & Agentic Ecosystem

In this section, we introduce the suite of specialized built-in data agents designed to handle complex workflows, and describe the broader agentic ecosystem that supports their customization and orchestration.

Data Agents: Besides the Conversational Analytics agents mentioned in Section 5 above, Data Cloud systems embed a number of other built-in data agents, designed to assist users with data-intensive tasks such as data science, data engineering, or data governance.

The *data engineering* agent[9] automates data preparation and data cleaning tasks. It supports complex data transformations and AI-assisted generation, modification, troubleshooting, and optimization of data pipelines.

The *data governance* agent was designed to assist with proactive governance: instead of waiting for compliance gaps, the system actively suggests classification, access controls, lineage tracking, and policy updates. The data governance agent also carries out some of the tasks of the harvesting component of

[9]https://cloud.google.com/blog/products/data-analytics/exploring-the-data-engineering-agent-in-bigquery

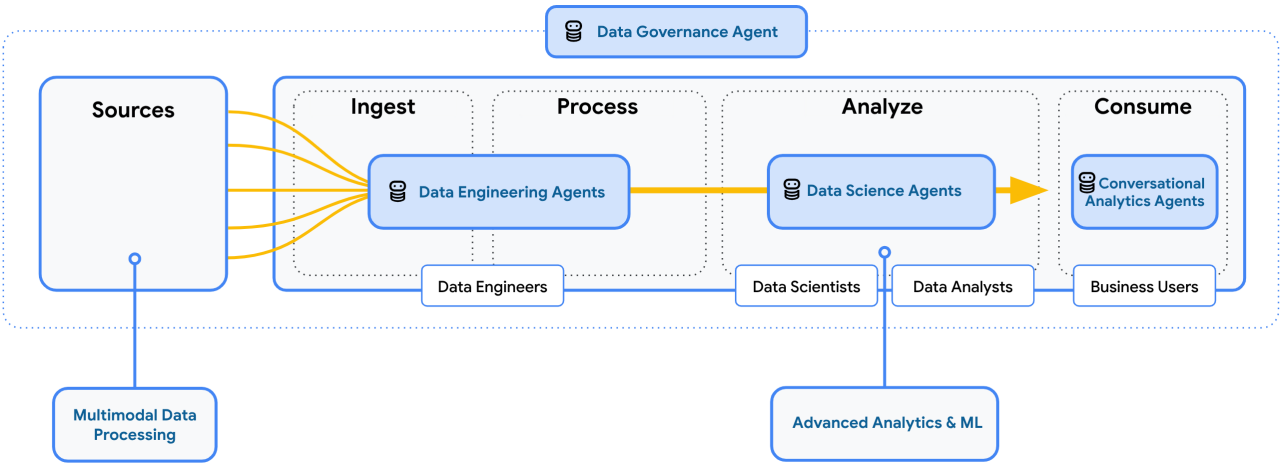


Figure 2: Built-in data agents in Google Data Cloud.

the Dataplex catalog, described in Section 4 above. These include automatic metadata generation such as table and column descriptions, finding common join patterns in the query history, or profiling the data in support of *value linking*.

The *data science* agent[10] helps streamline manual and repetitive data science tasks. It supports exploratory data analysis, highlighting patterns in the data via rich visualizations, as well as development and evaluation of statistical, timeseries, and predictive ML models. The data science agent leverages the full power of Colab notebooks via Python code generation and makes use of the rich variety of libraries available in the Python ecosystem for data science, machine learning, and advanced visualizations. To enable scaling to very large datasets, the data science agent can generate PySpark data pipelines and automatically provision serverless Spark clusters for distributed parallel execution. Alternatively, the data science agent can generate BigFrames[11] code, which transpiles to SQL and executes on the scalable BigQuery execution engine. The data science agent can effectively generate BQML[12] code to both take advantage of advanced built-in models (e.g., TimesFM for time series analysis and forecasting) as well as train custom models from the user's datasets and use them for inference. Improving the quality of the data science agent is an area of active research[44].

Agentic Ecosystem: The true power of any agentic ecosystem lies in its extensibility, allowing users to deeply integrate and customize their own data agents alongside the platform's built-in offerings. Users can tap into this ecosystem through several key components:

- *Custom data agent creation & hosting:* Developers can build their own fully customized data agents using the Agent Development Kit (ADK[13]) and the Antigravity[14] ecosystem, and they can deploy, manage, and scale agents in production with Agent Engine[15]. A low-code alternative to developing custom data agents, is to package specialized context and system instructions in custom Conversational Analytics agents using the control plane API introduced in section 5 above (figure 1b).

[10] https://cloud.google.com/blog/products/ai-machine-learning/ai-first-colab-notebooks-in-bigquery-and-vertex-ai
[11] https://docs.cloud.google.com/bigquery/docs/bigquery-dataframes-introduction
[12] https://docs.cloud.google.com/bigquery/docs/bqml-introduction
[13] https://google.github.io/adk-docs
[14] https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform
[15] https://docs.cloud.google.com/agent-builder/agent-engine/overview

- *Tool integration:* Developers can equip their custom agents with Data Cloud functionalities by utilizing the MCP (Model Context Protocol) Toolbox for Databases[16], which provides a suite of prebuilt and custom tools that enable AI agents to securely and efficiently interact with over 40 different databases, including AlloyDB, Cloud SQL, Spanner, BigQuery, MySQL, PostgreSQL, Neo4j, MongoDB, and more.

- *Agent delegation:* Through the A2A[17] (Agent-to-Agent) protocol, users can connect their custom agents to Data Cloud's built-in specialized agents, allowing them to smoothly delegate complex data tasks. Looking forward, this highly interoperable ecosystem paves the way for complex agent meshes and networks, where built-in Data Cloud agents can seamlessly act as specialized sub-agents for custom data agents across the enterprise. Effective and efficient orchestration of multi-agent systems is an active area of research[45–50].

While disambiguation capabilities help agents clarify user requests, repeatedly forcing users to answer the same clarifying questions can quickly become frustrating. To mitigate this, agentic data systems must learn from their user interactions over time, thus *memory* is a critical mechanism for all data agents[51]. Crucially, the system synthesizes the semantic context gleaned from individual user conversations and makes it available to all users within a customer's organization. This shared organizational memory ensures that new users, or those who are less experienced with the specific nuances of the company's data estate, are effectively assisted from their very first interaction, eliminating the cold start problem.

An area of current and future work regards enhancing autonomous decision-making via specialized agentic skills[18]. Realizing this vision, however, introduces several significant technical challenges, particularly in how skills are autonomously discovered, composed, and continuously updated. Recent research has increasingly focused on dynamic skill acquisition and hierarchical reasoning trees to move beyond static, handcrafted workflows; yet, elevating LLM capabilities from simple API tool-calling to the robust composition of generalized, reusable skills remains a complex hurdle [52]. Furthermore, automating the discovery of these new skills in open-ended environments—without relying on predefined task boundaries or human demonstrations—poses a major exploration challenge that recent systems attempt to address through exploration-first strategies [53]. Additionally, there is the ongoing technical challenge of continuously evolving and self-improving these skill libraries, with active research exploring recursive reinforcement learning methods to iteratively build and refine more complex agentic behaviors over time [54].

Within an enterprise data ecosystem, these challenges extend to accurately grounding these skills in complex data environments. Using the enterprise catalog's knowledge graph, the system could create semantic mappings between unstructured or tabular data assets and specific recipes for performing common tasks on these datasets. However, the remaining technical difficulty lies in enabling agents to dynamically and selectively retrieve and inject only the most relevant task recipes or playbooks directly into their limited context window, which is critical for significantly improving task reliability without degrading model performance.

7 Conclusion

In this paper, we present several foundational components of Google's AI-powered Data Cloud architecture. In particular, the AI functions, the modern search stack, universal catalog and semantic models are core building blocks for developing intelligent agents and AI applications. In this agents-oriented system,

[16] https://github.com/googleapis/genai-toolbox
[17] https://developers.googleblog.com/developers-guide-to-ai-agent-protocols
[18] https://agentskills.io/home

natural language is the new way to interact with the services, while the system complexities are expected to be hidden from the customers. We believe the future will be agentic and the ease of use will be a key differentiator.

In this paper, we also identify several research challenges that need to be addressed for this vision. AI functions enable operational databases and analytical platforms to extract information from structured, unstructured and multimodal data and execute deep analytics and powerful semantic search. We presented our list of new AI functions, but we believe there are opportunities for new functions packaging LLM abilities in novel ways that compose better and unlock new use-cases. Each of these new functions can benefit from novel implementations that use the LLMs in most efficient ways. Moreover, these new functions, and mixing them with relational operations, further complicate query optimization. The optimization problem is no longer just minimizing latency, but also maximizing accuracy. This new dimension creates a pareto frontier that requires new optimizers [55]. The optimization of the quality/cost tradeoff is particularly accurate for search that combines structured data and unstructured/multimodal data. The modern search funnel includes ANN vector search as its low cost / medium quality layer and AI Functions for filtering and ranking as the medium cost or high cost, but also high quality, layer. Furthermore, it is important that we make SQL (as) declarative (as possible) again and get rid of the knobs [56].

Understanding the data semantics and business logic remains the main technical challenge for NL2X and conversational analytics agents. Universal catalogs and semantic models that capture these enterprise-specific information is critical to map the user intent into the correct set of tables and columns, and identify correct literals to use in predicates. How to collect and organize the most effective metadata, and how to reason about it efficiently at scale are still challenging problems. While models have evolved to correctly generate standard SQL found in many benchmarks, they need additional context, and tuning to generate recent SQL extensions, including vector search queries, graph queries and queries with AI functions.

Agents are transforming the enterprise architectures to solve many enterprise data problems, including data engineering tasks, data science tasks, and data governance tasks. For all data agents, skills are emerging as the new mechanism to solve many subtasks. Figuring out how and when to use skills effectively and efficiency is an open challenge. While data science agents are solving critical problems, they still have room for improving their accuracy. Finally, many agentic solutions are composed of multiple agents, and effective and efficient orchestration of multi-agent systems is an area that warrants more research.

References

[1] F. Özcan and Y. Chung and Y. Chronis and Y. Gan and Y. Wang and C. Binnig and J. Wehrstein and G. Kakkar and S. Abu-el-haija, "LLMs and Databases: A Synergistic Approach to Data Utilization," *IEEE Data Eng. Bull.*, vol. 49, no. 1, pp. 32–44, 2025.

[2] S. Mishra and Y. Papakonstantinou, "Alloydb ai drives innovation for application developers," Google Cloud Blog, April 2025, accessed: 2026-02-17. [Online]. Available: https://cloud.google.com/blog/products/databases/alloydb-ai-drives-innovation-from-the-database

[3] A. Verma and J. Burr. (2024, Apr.) SQL reimagined for the AI era with BigQuery AI functions. Google Cloud Blog. Accessed: Feb. 27, 2025. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/sql-reimagined-for-the-ai-era-with-bigquery-ai-functions

[4] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *International*

Conference on Learning Representations (ICLR), 2023, foundational for reasoning and tool use integration.

[5] T. Schick *et al.*, "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023, details how LLMs learn to call APIs for external execution.

[6] S. Liu, S. Ponnapalli, S. Shankar, S. Zeighami, A. Zhu, S. Agarwal, R. Chen, S. Suwito, S. Yuan, I. Stoica *et al.*, "Supporting our ai overlords: Redesigning data systems to be agent-first," *arXiv preprint arXiv:2509.00997*, 2025.

[7] Y. Zhu, L. Wang, C. Yang, X. Lin, B. Li, W. Zhou, X. Liu, Z. Peng, T. Luo, Y. Li *et al.*, "A survey of data agents: Emerging paradigm or overstated hype?" *arXiv preprint arXiv:2510.23587*, 2025.

[8] M. Rahman, A. Bhuiyan, M. S. Islam, M. T. R. Laskar, R. Mahbub, A. Masry, S. Joty, and E. Hoque, "Llm-based data science agents: A survey of capabilities, challenges, and future directions," *arXiv preprint arXiv:2510.04023*, 2025.

[9] P. Liskowski, B. Han, P. Aggarwal, B. Chen, B. Jiang, N. Jindal, Z. Li, A. Lin, K. Schmaus, J. Tayade *et al.*, "Cortex aisql: A production sql engine for unstructured data," *arXiv preprint arXiv:2511.07663*, 2025.

[10] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE intelligent systems*, vol. 24, no. 2, pp. 8–12, 2009.

[11] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li *et al.*, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025.

[12] L. Mei, J. Yao, Y. Ge, Y. Wang, B. Bi, Y. Cai, J. Liu, M. Li, Z.-Z. Li, D. Zhang *et al.*, "A survey of context engineering for large language models," *arXiv preprint arXiv:2507.13334*, 2025.

[13] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3580–3599, 2024.

[14] Google Cloud, "What is AlloyDB AI?" Google Cloud Documentation, February 2026, accessed: 2026-02-17. [Online]. Available: https://docs.cloud.google.com/alloydb/docs/ai/what-is-alloydb-ai

[15] R. D. Amit Ganesh. (2025, Feb.) Powering the next generation of agents with google cloud databases. Google Cloud Blog. Describes the integration of Model Context Protocol (MCP) to standardize agent-database interactions. [Online]. Available: https://cloud.google.com/blog/products/databases/managed-mcp-servers-for-google-cloud-databases

[16] Google Antigravity Team. (2025, Nov.) Build with Google Antigravity, our new agentic development platform. Google Developers Blog. Announcement of a task-oriented IDE designed for autonomous agent orchestration. [Online]. Available: https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform/

[17] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia, "Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus," *Proc. VLDB Endow.*, 2025. [Online]. Available: https://doi.org/10.14778/3749646.3749685

[18] C. Liu, M. Russo, M. J. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. J. Franklin, T. Kraska, S. Madden, R. Shahout, and G. Vitagliano, "Palimpzest: Optimizing ai-powered analytics with declarative query processing," in *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025. [Online]. Available: https://vldb.org/cidrdb/2025/palimpzest-optimizing-ai-powered-analytics-with-declarative-query-processing.html

[19] J. Zhang and et al., "Docetl: Agentic query rewriting and evaluation for complex document processing," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3035–3048, 2025. [Online]. Available: https://arxiv.org/abs/2410.12189

[20] S. Jo and I. Trummer, "Thalamusdb: Approximate query processing on multi-modal data," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 186, 2024. [Online]. Available: https://doi.org/10.1145/3654989

[21] Y. Chung, R. Desai, J. He, Y. Xiao, T. Hottelier, Y.-L. K. Samo, P. Kadilkar, X. Chen, S. Idicula, F. Özcan, A. Halevy, and Y. Papakonstantinou, "100x cost & latency reduction: Performance analysis of ai query approximation using lightweight proxy models," https://arxiv.org/abs/2603.15970, 2026, accepted at SIGMOD 2026, Bengaluru, India.

[22] S. Zeighami, S. Shankar, and A. Parameswaran, "Featurized-decomposition join: Low-cost semantic joins with guarantees," *arXiv preprint arXiv:2512.05399*, 2025.

[23] I. Trummer, "Implementing semantic join operators efficiently," *arXiv preprint:2510.08489*, 2026.

[24] J. Lao, A. Zimmerer, O. Ovcharenko, T. Cong, M. Russo, G. Vitagliano, M. Cochez, F. Özcan, G. Gupta, T. Hottelier *et al.*, "Sembench: A benchmark for semantic query processing engines," *arXiv preprint arXiv:2511.01716*, 2025.

[25] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, 2020.

[26] D. Lu, H. Caminal, M. Chatzakis, Y. Papakonstantinou, Y. Chronis, V. Jain, and F. Özcan, "An in-depth study of filter-agnostic vector search on a postgresql database system," https://arxiv.org/abs/2603.23710, 2026, accepted at SIGMOD 2026, Bengaluru, India.

[27] A. Kusupati and et.al., "Matryoshka representation learning," 2022. [Online]. Available: https://arxiv.org/abs/2205.13147

[28] M. Stonebraker, I. F. Ilyas *et al.*, "Data integration: The current status and the way forward." *IEEE Data Eng. Bull.*, vol. 41, no. 2, pp. 3–9, 2018.

[29] A. Doan, A. Y. Halevy, and Z. G. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012. [Online]. Available: http://research.cs.wisc.edu/dibook/

[30] A. Quamar, V. Efthymiou, C. Lei, and F. Özcan, "Natural language interfaces to data," *Found. Trends Databases*, vol. 11, no. 4, pp. 319–414, 2022. [Online]. Available: https://doi.org/10.1561/1900000078

[31] G. Katsogiannis-Meimarakis, M. Xydas, and G. Koutrika, "Natural language interfaces for databases with deep learning," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3878–3881, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p3878-katsogiannis-meimarakis.pdf

[32] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Özcan, and S. O. Arik, "CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2025. [Online]. Available: https://openreview.net/forum?id=CvGqMD5OtX

[33] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[34] Y. Chung, G. T. Kakkar, Y. Gan, B. Milne, and F. Özcan, "Is long context all you need? leveraging llm's extended context for nl2sql," *Proceedings of the VLDB Endowment*, vol. 18, no. 8, p. 2735–2747, Apr. 2025. [Online]. Available: http://dx.doi.org/10.14778/3742728.3742761

[35] N. Huo, X. Xu, J. Li, P. Jacobsson, S. Lin, B. Qin, B. Hui, X. Li, G. Qu, S. Si *et al.*, "Bird-interact: Re-imagining text-to-sql evaluation for large language models via lens of dynamic interactions," *arXiv preprint arXiv:2510.05318*, 2025.

[36] J. Nam, J. Yoon, J. Chen, J. Shin, S. O. Arık, and T. Pfister, "Mle-star: Machine learning engineering agent via search and targeted refinement," 2025. [Online]. Available: https://arxiv.org/abs/2506.15692

[37] P. Yin, W. Ding, C. Xia, L. Wang, C. Carroll, J. Li, G. Neubig *et al.*, "Natural language to code generation in interactive data science notebooks," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 141–173.

[38] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[39] Z. Wang, Z. Zhou, D. Song, Y. Huang, S. Chen, L. Ma, and T. Zhang, "Towards understanding the characteristics of code generation errors made by large language models," 2025. [Online]. Available: https://arxiv.org/abs/2406.08731

[40] J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan *et al.*, "Mle-bench: Evaluating machine learning agents on machine learning engineering," *arXiv preprint arXiv:2410.07095*, 2024.

[41] F. Shu, Y. Wang, R. Wu, B. Liu, Z. Yao, Y. He, and F. Yan, "Dare-bench: Evaluating modeling and instruction fidelity of llms in data science," 2026. [Online]. Available: https://arxiv.org/abs/2602.24288

[42] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.

[43] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 481–503, 2025.

[44] J. Nam, J. Yoon, J. Chen, R. Sinha, J. Shin, and T. Pfister, "DS-STAR: Data science agent for solving diverse tasks across heterogeneous formats and open-ended queries," *arXiv preprint arXiv:2509.21825v4*, 2026.

[45] A. Adimulam, R. Gupta, and S. Kumar, "The orchestration of multi-agent systems: Architectures, protocols, and enterprise adoption," 2026. [Online]. Available: https://arxiv.org/abs/2601.13671

[46] P. Drammeh, "Multi-agent llm orchestration achieves deterministic, high-quality incident response," 2025. [Online]. Available: https://arxiv.org/abs/2511.15755

[47] G. Yu, "Adaptorch: Task-adaptive multi-agent orchestration in the era of performance convergence," 2026. [Online]. Available: https://arxiv.org/abs/2602.16873

[48] Y. Dang, C. Qian, X. Luo, J. Fan, Z. Xie, R. Shi, W. Chen, C. Yang, X. Che, Y. Tian, X. Xiong, L. Han, and M. Sun, "Multi-agent collaboration via evolving orchestration," 2025. [Online]. Available: https://arxiv.org/abs/2505.19591

[49] W. Zhang, L. Zeng, Y. Xiao, Y. Li, C. Cui, Y. Zhao, R. Hu, Y. Liu, Y. Zhou, and B. An, "Agentorchestra: Orchestrating multi-agent intelligence with the tool-evolutionary agent (tea) protocol," 2025. [Online]. Available: https://arxiv.org/abs/2506.12508

[50] K. Agrawal and N. Nargund, "Neural orchestration for multi-agent systems: A deep learning framework for optimal agent selection in multi-domain task environments," 2025. [Online]. Available: https://arxiv.org/abs/2505.02861

[51] S. Ouyang, J. Yan, I.-H. Hsu, Y. Chen, K. Jiang, Z. Wang, R. Han, L. T. Le, S. Daruki, X. Tang, V. Tirumalasetty, G. Lee, M. Rofouei, H. Lin, J. Han, C.-Y. Lee, and T. Pfister, "ReasoningBank: Scaling agent self-evolving with reasoning memory," *arXiv preprint arXiv:2509.25140*, 2025, accepted to ICLR 2026. [Online]. Available: https://arxiv.org/abs/2509.25140

[52] R. Xu *et al.*, "SoK: Agentic skills – beyond tool use in LLM agents," *arXiv preprint arXiv:2602.20867*, 2026. [Online]. Available: https://arxiv.org/abs/2602.20867

[53] A. Authors, "EXIF: Automated skill discovery through exploration-first strategy," *arXiv preprint arXiv:2506.04287*, 2025. [Online]. Available: https://arxiv.org

[54] Y. Wang *et al.*, "SkillRL: Evolving agents via recursive skill-augmented reinforcement learning," *arXiv preprint arXiv:2602.08234*, 2026. [Online]. Available: https://arxiv.org

[55] M. Russo, S. Sudhir, G. Vitagliano, C. Liu, T. Kraska, S. Madden, and M. Cafarella, "Abacus: A cost-based optimizer for semantic operator systems," *arXiv preprint arXiv:2505.14661*, 2025.

[56] M. Chatzakis, Y. Papakonstantinou, and T. Palpanas, "DARTH: declarative recall through early termination for approximate nearest neighbor search," *Proc. ACM Manag. Data*, vol. 3, no. 4, pp. 242:1–242:26, 2025. [Online]. Available: https://doi.org/10.1145/3749160

The Duality between Large Language Models and Data Management Systems

M. Tamer Özsu and Kerem Akillioglu and Xiangru Jian

Abstract

The interaction between large language models (LLMs) and data management systems (DMSs) has emerged as one of the most consequential intersections in contemporary computer science. This interaction is bidirectional. In one direction (LLM4DB), LLMs assist, augment, or replace components of a DMS. Cooperation with LLMs facilitates tasks such as natural language querying, data cleaning and transformation, source integration, and semantic annotation. In the other direction (DB4LLM), DMS architectures and techniques support the management and preparation of managing large-scale multimodal training datasets for LLMs. They enable low-latency approximate nearest neighbour search over high-dimensional vector data for retrieval-augmented generation at inference time. These two directions exhibit a duality in which advances on one side create demand and opportunity on the other. In this paper, we report on research on both sides of this duality. In this paper, we discuss two projects in this space that we are engaged in, one in each direction.

1 Introduction

There is significant interest, both in academia and in industry, in investigating the alignment between large language models (LLMs) and data management systems (DMS)[1]. Some share the view that LLMs call into question the very reasons for the existence of the DMSs that centre around how to answer questions over data accurately and efficiently [17]. Others argue that LLMs can address the limitations of "traditional" machine learning (ML) models in data management and claim that LLMs offer opportunities throughout the data pipeline [71]. These discussions are addressing one direction of the interaction: LLMs for DMSs (LLM4DB), but the interaction is bidirectional, and there is a strong case to be made for data management technology assisting LLMs (DB4LLM). These two directions exhibit a duality where each strengthens (or can strengthen) the other.

The first direction (DB4LLM) addresses how DMS architectures and techniques can be used for the training, serving, and retrieval requirements of the LLMs. Modern LLMs are data-intensive systems, and their performance (in terms of quality) is tightly bound by the training dataset. The quality, coverage, and organization of this dataset are, therefore, a significant concern. Data engineering provides a systematic way of ensuring that the dataset is of appropriate quality through appropriate data management techniques and the data preparation pipeline. The datasets that LLMs use are large-scale, multimodal data, and the management of this data is what the data management community has expertise in. The argument is not to use the existing technologies (e.g., relational DBMSs) but the technologies that have been developed over the years (e.g., parallel data management, geo-distributed data management, NoSQL system technologies, and stream data processing). The second component of this direction is data preparation, which is the process of data acquisition, data integration, dataset

[1]We use the term "data management system" (DMS) to include systems that may not have full database management system (DBMS) functionality, such as the NoSQL systems

selection, and data quality enforcement. Data preparation ensures that the training datasets that LLMs use are of high quality [3]. A recent challenge in this direction is the incorporation into the training dataset of very high-dimensional vector data through embeddings with a new workload: approximate nearest neighbour (ANN) search. On the serving side, retrieval-augmented generation (RAG) combines LLM inference with an external knowledge store that is queried at runtime[31]. More specifically, for reasonable latency, RAG systems demand low-latency approximate nearest neighbour (ANN) search over billions of indexed vectors from multimodal data[46]. The quality of LLM-generated responses is fundamentally determined by the performance characteristics of this retrieval layer.

The second direction (LLM4DB) reverses the connection: it concerns the use of LLMs to assist, augment, or replace components of a DMS. The central point of this direction is that LLMs can assist across a wide range of DMS tasks: translating natural language into executable queries, cleaning and transforming messy data, integrating heterogeneous sources, generating semantic annotations, and supporting decision-making. In these ways, they lower the expertise barrier for interacting with complex DMSs. One of these tasks, namely the access of databases in natural language, is a long-term ambition of the database community, with early work going back to Codd [11] and with repeated attempts subsequently (e.g., [21, 32, 50, 63, 70]). This objective proved generally elusive until LLMs arrived with sufficient language understanding to make it practically viable. Today, LLM4DB initiatives are broader, and a number of major vendors embed LLM-based operators directly within SQL. This allows queries to invoke model inference on data as a first-class query operation [2].

These two directions represent the duality between them. This paper reports on our research situated on both sides. In DB4LLM direction, we describe our project that targets designing a data management stack that can support AI workloads. In LLM4DB direction, the project involves developing a multi-agent system to support natural language access to federated data sources.

2 Designing an AI-Native Data Management Stack

Providing a tighter integration of LLMs and database systems represents a new frontier in data infrastructure, one that requires solving problems beyond the scope of traditional database research. LLMs are trained on large amounts of data and are highly adaptable across downstream data management tasks such as natural language to SQL translation [58], data cleaning and integration [42], and semantic data processing [12, 25, 36, 47, 54], thereby enabling capabilities that are difficult or impossible to realize with conventional methods alone. This adaptability has already motivated direct integration of LLMs into SQL queries, also known as AI-SQL, as seen in PostgreSQL and DuckDB extensions [15, 61], as well as in offerings from major data platforms such as Google [18], Snowflake [35], and Databricks [13]. Another increasingly important use case is the emergence of LLM-based data agents that autonomously query and interact with data systems to automate complex, multi-step workflows [41]. This marks a shift from using LLMs merely as components within data processing pipelines to treating LLM agents as users of data systems [38].

However, current data systems are not yet designed for emerging LLM-agent users. The existing DMS stack, from query languages and optimizers to indexes and resource managers, continues to assume a predictable user in the background, whereas LLMs and LLM-based data agents often exhibit longer and more variable execution times, irregular invocation patterns, and inherently non-deterministic behaviour. This is important because the mismatch is fundamental rather than incidental: it challenges the assumptions under which existing data systems expose abstractions, optimize execution, allocate resources, and maintain correctness. If data systems continue to be built around assumptions that no longer hold, they risk becoming a bottleneck for emerging AI-native applications, leading to inefficiency, higher cost, increased latency, and weaker reliability. Addressing this mismatch is therefore necessary to

ensure that data systems remain an effective foundation for LLM-agent workloads. We therefore aim to redesign data systems to better support the workloads generated by LLM agents.

The problem with most of the existing LLM-database integrations is that they use the existing DMS backends and make the optimizations to accommodate LLM or LLM-based agent workloads, which leads to further issues later on. In practice, this means current solutions use existing DMS and build *scaffolding* around the models of the moment through carefully engineered prompts, model-specific retrieval strategies, and pipeline configurations tuned to the strengths and weaknesses of a particular LLM. Yet such scaffolding can be fragile, and its optimizations do not reliably transfer across model upgrades or even across model families. Empirical studies on prompt robustness show that even meaning-preserving changes such as prompt formatting, instruction paraphrases, or the presentation of few-shot examples that can induce large performance swings, and prompt variants that work best for one model may correlate only weakly with those that work best for another [53]. Overall, model-tailored workflows face even greater generalization risks when the underlying model is replaced by a newer, stronger, or higher-reasoning one.

By contrast, the literature increasingly suggests that improvements in the *systems layer* and in the design of tools for LLMs [64] are more likely to remain useful as new models emerge. Recent work has explored large-scale tool-use infrastructure [48], context management and validation mechanisms for agentic workflows [8]. These contributions do not depend on a single prompt format or on compensating for the weaknesses of one particular model; instead, they provide better interfaces, better execution strategies, and better mechanisms for supplying and organizing information. When such tools are available, LLMs can reason about which tool is most appropriate for the task at hand. As a result, building better DMSs and tools for LLMs and LLM agents is more likely to complement stronger future models rather than be invalidated by them. This distinction motivates our focus: rather than investing primarily in better scaffolding around today's models, we argue for building better systems and tools *for* LLMs and LLM agents, so that the infrastructure becomes more valuable as the underlying models improve.

2.1 Workload Characteristics

Modern use cases increasingly require querying across heterogeneous systems and data sources [29]. A single workflow may need to access both structured and unstructured stores, such as relational databases, document stores, APIs, knowledge graphs, and unstructured file collections.

2.1.1 Read Operations

Read and write operations expose the mismatch between traditional data-system assumptions and LLM-agent behaviour in fundamentally different ways. On the read side, the mismatch lies between the exact structural assumptions of traditional data systems and the greater tolerance of LLM agents for ambiguity. In conventional systems, schema serves two main purposes: to preserve integrity under update (cf. normal forms), and to support efficient query processing and integrity enforcement. When the consumer is an LLM performing read-and-synthesize workloads, rather than executing precise analytical queries or maintaining referential integrity, both justifications weaken. LLMs are trained over enormous corpora of messy, weakly aligned, and heterogeneous data, and thus have strong priors for resolving ambiguity across loosely structured sources. This resembles classical schema integration, but with a key difference: the consumer itself is semantically flexible, shifting part of the reconciliation burden from middleware to the model. As a result, systems may be able to relax some conventional schema assumptions for read operations. At the same time, this flexibility complicates optimization, since cost models must account not only for data access cost, but also for semantic interpretation, token

expenditure, and the reliability of model-mediated retrieval.

2.1.2 Write Operations

Write operations are less forgiving. When multiple LLM agents operate concurrently over shared mutable state, coordination and consistency cannot be left to the agents themselves. Unlike read-side ambiguity, where the model can often recover a plausible interpretation, concurrent writes expose the system to classical anomalies: lost updates when two agents overwrite the same artifact without awareness of one another, stale or dirty reads when one agent acts on partially updated state, and violations of causal ordering when logically dependent actions execute against inconsistent snapshots. These are familiar problems in database systems, but they become novel in agentic settings because agent outputs are non-deterministic, write intentions are harder to predict in advance, and multi-step actions may span both structured and unstructured states.

The central design question is therefore not whether coordination is necessary, but when it is necessary and what form it should take. Some classes of agent-written state admit coordination-free operation: append-only logs, monotonically growing sets, or last-writer-wins registers for non-critical metadata may tolerate weak consistency or well-defined merge semantics. Other classes of state require stronger guarantees, but the required strength depends on the property of interest: local constraint preservation, causal consistency, or full transactional isolation. This suggests that agent-oriented systems should expose a richer space of write semantics than the traditional binary choice between strict transactions and weak eventual consistency.

One promising point in this design space is optimistic concurrency control (OCC). Rather than synchronizing all agent actions eagerly, the system may allow speculative progress and validate updates at commit time. This is attractive in agentic settings because many concurrent actions will not conflict, while eager coordination may impose unnecessary latency and serialization. Yet OCC alone may be insufficient when writes are uncertain, semantically ambiguous, or difficult to validate automatically. In such cases, branching becomes a useful generalization of optimism: instead of forcing uncertain or potentially conflicting updates directly into shared state, the system can isolate them into separate branches for later validation, comparison, merging, or approval. In this sense, branching can be viewed as an extension of OCC for agentic environments, one that accommodates not only conflict detection, but also deferred semantic reconciliation and human oversight where needed.

For state that does not require strict isolation, but for which coordination-free convergence is too weak, probabilistic consistency models offer a principled middle ground. Probabilistically Bounded Staleness (PBS) [5] showed that eventually consistent partial-quorum systems often return consistent data within milliseconds of a write, and that staleness can be quantified continuously along both version and time axes rather than treated as a binary property. This perspective is especially appealing for agentic workloads, where some reads—for example, over context summaries, cached tool outputs, or non-critical metadata—may tolerate bounded staleness in exchange for lower latency. PBS therefore provides a useful foundation for reasoning about freshness-latency trade-offs in LLM-facing systems. However, the original PBS analysis focused on single-key staleness. Extending such reasoning to the multi-key, multi-agent setting required by agentic architectures, where causal relationships and atomicity requirements span multiple objects and actions, remains an open problem.

2.2 Deep Dive into the Stack

Redesigning data systems for LLM-agent users brings challenges with little precedent in traditional database research and calls for a fundamental rethinking of the data stack. This requires not only revisiting existing components, but also introducing new ones tailored to the demands of these emerging

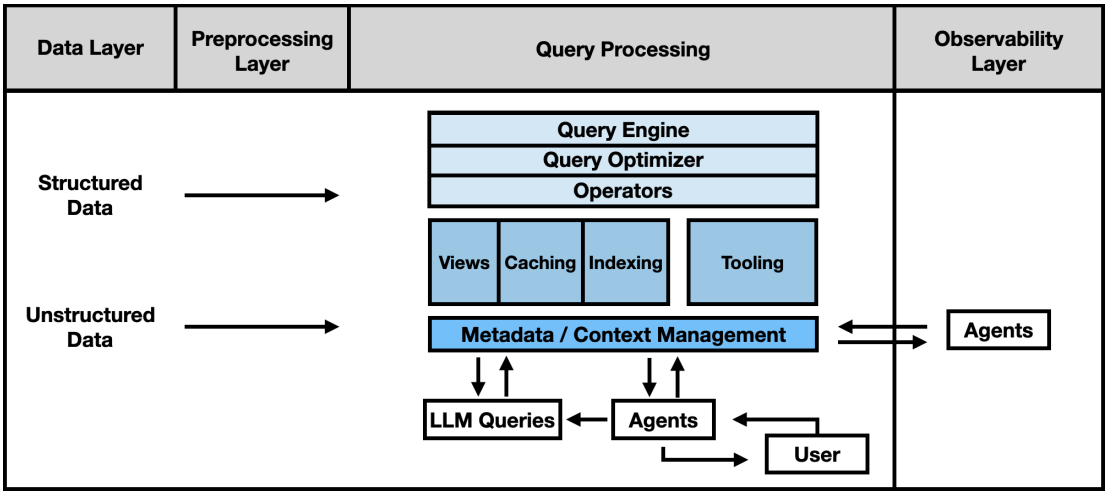


Figure 1: AI Native Stack Architecture.

workloads. Across these design choices, three key dimensions are worth optimizing for: accuracy, cost, and system efficiency. Figure 1 illustrates our overall architecture. In this section, we discuss the key aspects of workload characteristics, system's querying interface, the adaptations needed in the existing DMS architecture, and the new components required to support the LLM-agent workload.

2.2.1 Querying Interface

Heterogeneous querying naturally raises the question of how query interfaces should be reconsidered for the new system. A structured declarative interface is particularly helpful when large or complex queries are difficult to express reliably in natural language alone. At the same time, even as workloads change, much of the world's valuable data remains stored in relational DBMSs. For that reason, we emphasize that nearly any system developed in this space should continue to support SQL, given its longevity and central role in data management [59]. A natural direction, then, is to build on top of SQL rather than discard it entirely. AI-SQL syntax can be viewed in this light, as a more expressive interface that extends SQL to better support LLM functionality, heterogeneous data access, and agent-driven workloads while preserving a declarative style. However, the implications of these new workloads for query languages and interfaces go beyond simply extending SQL. Prior critiques have already pointed out that SQL's irregular syntax, implicit ordering between clauses, and limited abstraction mechanisms create usability challenges even for human users [43]. For LLM agents, these issues become even more pronounced: irregular grammar increases the error surface for query generation, verbosity inflates token costs at scale, and fragmentation into incompatible dialects across systems makes querying heterogeneous sources especially fragile. This suggests value not only in supporting SQL and exploring extensions such as AI-SQL, but also in investigating alternative representations and interfaces that may reduce token cost, improve robustness, and better fit multi-source, agent-oriented workloads.

2.2.2 Rethinking Existing Components and Building New Ones

Building views, caching, and indexing will remain central, but they need to be reconsidered for LLM-agent workloads. Classical work has already shown how powerful views can be for query answering, source integration, and reusing expensive computation across queries [20]; in the same spirit, an AI-native stack should revisit these mechanisms under the assumptions of semantic, multi-step, and heterogeneous workloads. A view may no longer be just a stored relational subquery, but also a reusable

intermediate result such as a filtered context set, an extracted schema summary, or a semantically enriched representation prepared for downstream reasoning. Caching similarly extends beyond tuples and pages to model responses, prompt-conditioned retrieval results, and intermediate outputs whose reuse depends not only on key equality, but also on task context, approximation tolerance, and model choice. Indexing, in turn, must support not only exact predicate evaluation, but also semantic access paths over embeddings, metadata, and cross-source references. The point is that the role of traditional components expands: they must now help the system reduce repeated model calls, control token and compute cost, and support more efficient and reliable execution over heterogeneous data and agent-driven workflows.

Managing context becomes a first-class systems problem in our setting. Unlike a traditional query processor, an agent repeatedly invokes an LLM across multiple steps, and each step must decide what information from prior interactions, retrieved data, tool outputs, and task instructions should be carried forward into the next call. This makes context construction far more than a prompt-writing issue: it requires dynamically selecting, compressing, and structuring the right state for the current decision. Prior work already shows that seemingly small changes in prompt wording or example selection can materially affect performance [42, 49], while irrelevant information in the input can degrade quality even when the relevant information is present [57]. For agentic systems, this implies that context should not be treated as a raw transcript or a full dump of available state, but as a curated representation of only the information necessary for the next action. A system that exposes too little context leaves the agent under-informed; a system that exposes too much increases token cost, dilutes attention, and raises the likelihood of error.

Metadata management is equally important because agents often rely on auxiliary information to interpret data, select tools, and decide what to do next. Schema descriptions, data dictionaries, tool specifications, retrieved examples, execution history, and summaries of prior intermediate results can all improve performance when they are relevant, but they also introduce overhead and can interfere with reasoning when included indiscriminately. This tension becomes more severe in multi-step workflows, where mistakes in early context selection can compound over time. Studies of agentic search, for example, suggest that once an agent is nudged toward the wrong local path, it may continue refining that path rather than broadening its search, amplifying early errors across subsequent steps [44]. From this perspective, metadata is not just supporting information; it is part of the control surface of the system. The key challenge is therefore not merely to store metadata, but to determine what metadata should be surfaced, in what form, and at which step, so that the agent remains both well-grounded and efficient.

Tools are equally important in this setting because they shape how agents plan, decompose, and execute tasks over data systems. Examples include schema inspection tools that reveal available tables and attributes, query execution tools that allow the agent to test and refine candidate queries, vector retrieval tools that fetch relevant documents or rows, validation tools that check whether outputs satisfy constraints, and cost-estimation tools that help compare alternative plans. Consider an agent asked to "find customers whose recent support tickets mention billing issues and summarize the common causes." Without tools, the agent would have to guess the schema, write a query from memory, and reason about the results entirely within the model. With tools, it can first inspect the schema to discover the ticket and customer tables, issue a query to retrieve recent billing-related tickets, use retrieval or semantic filtering to identify relevant text, validate that the join keys and predicates are correct, and then summarize only the filtered results. In this way, tools do not merely assist execution; they help the agent form better plans by grounding each step in system feedback, reducing unnecessary reasoning, and making multi-step interaction with heterogeneous data more reliable.

Taken together, these considerations have direct implications for the design of operators, the query optimizer, and the execution engine. The system can no longer rely solely on traditional relational operators and cost assumptions, but must introduce operators that account for semantic retrieval, model invocation, validation, tool use, and interaction across heterogeneous sources. The optimizer, in turn,

must reason not only about compute and I/O, but also about token cost, latency, model selection, approximation, and the uncertainty introduced by non-deterministic components. The execution engine must support adaptive, multi-step execution in which later decisions may depend on intermediate model outputs, tool feedback, and runtime conditions. Supporting LLM-agent workloads is therefore not simply a matter of attaching models to an existing engine; it requires rethinking the core machinery that determines what operations are available, how plans are chosen, and how those plans are executed.

2.3 Query Processing and Optimization

For the systems layer beneath AI agents, the central query-processing challenge is not merely supporting larger datasets, but supporting repeated and expensive interaction with LLM-based users. Each agent request may trigger multiple rounds of retrieval, reasoning, validation, and tool invocation, making execution costlier, less regular, and harder to predict than in traditional data systems. As a result, the system is no longer optimizing only for fast execution of a well-specified query, but for dependable support of requests that unfold through uncertain and adaptive steps. This tightly couples cost, latency, and accuracy: improving accuracy may require richer context, additional verification, or stronger models, each of which increases both response time and system cost. The role of the system, then, is not merely to execute requests efficiently, but to manage these tradeoffs in a principled way.

A major source of this difficulty is that query processing is increasingly dominated by the AI layer rather than by conventional database execution. Retrieval, filtering, joins, and other database operations may still complete relatively quickly, while the surrounding LLM calls dominate end-to-end latency and monetary cost. One practical observation is that not every interaction requires the same level of reasoning power: simpler cases may be handled by cheaper models, while only harder cases justify the use of frontier models. This motivates mechanisms such as *model cascades*, which have been explored in adjacent settings and, more recently, in AI-SQL and LLM-powered data processing systems [4, 9, 27, 39, 47]. From the perspective of the underlying system, however, cascades are only one part of the solution. They reduce the cost of individual model decisions, but do not change the fact that the system may still need to support too many model invocations overall. Running models locally shifts this burden from API spending to infrastructure provisioning, but does not eliminate it; the bottleneck simply moves from the billing layer to the GPU layer [2]. Thus, the core problem is not only where inference runs, but how the system determines when inference is necessary at all.

This has important implications for jointly considering logical and physical optimizations [1]. Existing work has explored physical optimizations such as batching, prefix caching, KV-cache compression, and model selection [37, 52, 55], as well as logical optimizations such as semantic operators and rewrite strategies that alter what computation is performed and in what order [15, 25, 47, 54]. However, these examples are largely drawn from AI-SQL and LLM-powered data-processing systems rather than from systems designed natively around AI-agent users. For a system serving AI agents, these two levels cannot be treated independently. Decisions about rewriting, decomposition, or tool selection affect how many model calls are made and what physical optimization opportunities remain; conversely, knowledge of model cost, cacheability, and latency should influence which logical alternatives the system prefers. Query optimization therefore becomes inherently multi-objective: the system must balance monetary cost, end-to-end latency, and overall system efficiency, while also accounting for approximation and uncertainty. The rewrite space becomes correspondingly richer. Optimization is no longer limited to algebraic equivalences inside a query plan, but may involve rewriting an entire execution pipeline, changing the order of retrieval and filtering, deciding when to rely on symbolic execution instead of a model, selecting among cascades of different cost and capability, or replacing repeated reasoning with cached or precomputed results. In this sense, the design challenge is broader than extending an existing engine with LLM support. It requires a system that can jointly reason about logical and physical choices

for AI-agent workloads. To our knowledge, this agent-native systems perspective remains largely open in the current literature.

3 A Multi-Agent System for Analytics over Heterogeneous Federated Systems

The preceding section addresses the cost of invoking LLMs at the granularity of individual data points. A complementary problem arises in *analytical* workloads: complex queries that combine evidence from multiple autonomous data sources to support decision-making. We define this problem, characterize what makes it hard, explain why existing approaches do not solve it, and present a multi-agent architecture that addresses the gap.

3.1 Problem Definition

Modern data science projects involve multimodal data from many different sources. Structured facts reside in relational DBMSs, entities and relationships required for linkage and reasoning are maintained in knowledge graphs, and narrative context and business details are recorded in document corpora. These data are held and managed by autonomous systems, each retaining its own data model, query interface, and operational semantics. In the database literature, such a setting is a *heterogeneous federated system*: a collection of autonomous data sources that must be queried together without physical centralization [45, 56]. The defining properties are source autonomy, heterogeneity, and virtual rather than materialized integration [56]. Traditionally, such federations are accessed through dedicated wrappers that expose each source's capabilities, coordinated by a mediator that decomposes queries across them [45, 66]. We extend that model beyond database sources to include knowledge graphs and document collections alongside relational DBMSs. Two structural features of this federated setting are central. First, the data is *multimodal*. Second, the federation is *heterogeneous* not only across modalities but within them: two relational DBMSs may expose different SQL dialects and schemas; two document collections may differ in whether layout information is indexed. No global schema governs the federation, and the system must discover what each source can do at query time [45].

We consider *analytical workloads*: mostly read-only, decision-support queries over historical data that combine operators such as filtering, aggregation, comparison, and linkage to identify trends, generate reports, and derive insights. We extend traditional relational OLAP workloads to those that may draw on semi-structured and unstructured data and require reasoning that spans modality boundaries. Consider the query: *"Assess the credit risk exposure across our top 20 suppliers over the past fiscal year: correlate payment delays and outstanding receivables with corporate ownership structures to identify concentrated risk, and flag any supplier whose audited financial statements show deteriorating liquidity ratios or covenant breaches."* Answering this requires relational aggregation over procurement and accounts-payable records (payment timeliness, invoice aging, outstanding receivables); graph traversal over corporate ownership and cross-guarantee relationships, discovering, for instance, that three apparently independent suppliers are subsidiaries of the same parent entity, which concentrates risk; and layout-aware extraction from audited financial statements and credit rating reports, visually complex PDF documents containing multi-column financial tables and trend charts whose liquidity ratios, debt-to-equity figures, and auditor qualification notes are not available in any structured source.

No single source or modality suffices, and the difficulty is not merely additive. Four properties distinguish such complex analytical tasks from simpler cross-source lookups:

1. **Source involvement is query-dependent**: which sources are relevant cannot be determined before the query is issued, i.e., data localization is required for each query. In the example, the

supplier ownership graph becomes necessary only after the relational aggregation reveals which suppliers qualify.

2. **Reasoning trajectories are correlated**: each step may condition on the output of previous steps, so that errors or omissions at any stage compound rather than average out. The ownership resolution, for instance, depends on the set of suppliers identified by the initial aggregation.

3. **Sources may require iterative access**: intermediate results may reveal new information needs that require returning to a previously consulted source, or accessing a source that was not part of the original plan. The initial graph traversal may surface a parent entity whose financial statements must then be retrieved, triggering a second round of document extraction not foreseeable at planning time.

4. **Evidence must be synthesized across modalities with provenance**: the final answer must compose results whose representations, semantics, and granularity differ across sources, and present the composite result with a trace sufficient for a domain expert to verify each claim against its origin. The credit risk assessment must reconcile a numeric total from a relational table, an ownership path from a knowledge graph, and an auditor qualification from a PDF, each traceable to its source.

These properties make it impractical for users to manually formulate and execute such queries. Domain experts understand what they need to know, but cannot efficiently execute the multi-step, multi-paradigm procedures their questions require [26, 28]. Analytics teams face backlogs that span weeks or months, and many questions go unasked because the cost of formulating them exceeds the perceived payoff.

LLMs offer a path forward: their natural language understanding, code generation, and tool invocation capabilities make it possible to interact with diverse data sources on the user's behalf, bridging the gap between what domain experts can ask and what federated systems can execute. This work falls within the direction of using LLMs for data management, specifically using LLM-based agents to perform complex analytical tasks over heterogeneous federated systems that were previously infeasible without deep technical expertise across multiple query paradigms.

3.2 Why Existing Approaches Fall Short

Several existing approaches address parts of this problem, but none solves it fully.

Single-modality natural language interfaces. Text-to-SQL systems translate natural language into SQL [23, 40], KGQA systems generate SPARQL queries [30], and RAG systems retrieve passages and condition a language model on them [19, 31]. Each is confined to its modality: when an analytical task spans multiple modalities, no single-modality interface can serve as the sole solution.

Polystore and data federation systems. Polystore systems such as BigDAWG [16] and query frameworks such as Apache Calcite [6] provide execution infrastructure across multiple storage engines. However, cross-engine decomposition remains a user responsibility, they require formal query languages (which do not help domain experts who lack the technical skills to write them), they rely on upfront schema mediation that contradicts the virtual integration model of federated systems [56], and they primarily target structured data with limited support for unstructured documents.

LLM-based agents. LLM-based agents [69] can accept natural language and invoke external tools in sequence, but the four complexity properties expose fundamental limitations. Because sources differ in what they can do and the relevant sources are not known in advance, the agent must reason about each source's characteristics; in practice, single-agent systems treat all APIs as interchangeable [62]. Because reasoning trajectories are correlated, an error in an early step propagates through all subsequent steps

with no mechanism for intermediate validation. When combined with RAG, all data access is reduced to retrieving passages and then generating text, which cannot replace the structured computation that different sources provide [19]. The agent also typically produces a final answer without maintaining a record of which source contributed which claim.

General-purpose multi-agent frameworks. AutoGen [68], Camel [33] and MetaGPT [22] allow multiple agents to collaborate, but they are general-purpose coordination tools, not designed for heterogeneous data analytics. Agents decide what to do next through conversation without an explicit plan that can be validated before execution. The frameworks do not model what each data source can do, so source-level reasoning limitations carry over from single-agent designs. Intermediate results are exchanged as unstructured text without provenance records.

In sum, none of these approaches fully addresses the problem. To enable complex analytical tasks over heterogeneous federated systems, we need a dedicated multi-agent architecture in which each data source is served by a specialized agent that understands that source's specific characteristics, coordinated by a planning and execution layer that can reason about what each source can do, validate intermediate results, and assemble a traceable final answer. We refer to this coordination as *compositional orchestration*.

3.3 A Multi-Agent Architecture

The traditional mediator/wrapper approach [45, 66] works when queries are expressed in a formal language, the relevant sources can be identified from the schema, and decomposition follows deterministic rewriting rules. The four complexity properties require capabilities beyond static query processing: interpreting natural language input, determining which sources are relevant as intermediate results reveal new needs, evaluating partial results to decide whether to proceed or re-plan, and composing evidence across representations with no shared data model. These decisions depend on the content and quality of intermediate results at runtime and cannot be specified as rewriting rules before execution begins. What might work better is an *agentic* approach: system components that can autonomously perceive their operating context, reason about what action to take, and adapt based on observed outcomes [67].

The heterogeneity of sources, not just across modalities but within them, means that interacting with a given source requires understanding that source's specific interface, schema, and operational constraints. A single agent cannot maintain this depth across all sources in the federation. This motivates a *multi-agent* design in which each source is served by a dedicated specialist. Figure 2 shows the resulting architecture, which comprises five components and a provenance-by-construction design. The upper layer contains three *general agents* (User Agent, Orchestrator, Executor) that handle query refinement, planning, and execution coordination, all grounded in a shared Metadata Repository (MR). The lower layer contains *Specialized Data Agents* (SDAs), each dedicated to a specific data source and interacting with it via native query execution. Provenance records flow through the entire pipeline, from individual SDA results back through the Executor to the final answer presented by the User Agent.

Specialized Data Agents (SDAs). Each data source is served by a dedicated agent operating within a purpose-built *execution environment* that determines which operations the agent can perform. We call the complete set of these operations the source's *action space*. The action space includes all operations exposed by the source's native query interface as well as any prebuilt operations for frequently needed tasks. For this prototype, we develop an SDA for relational DBMSs whose action space covers the full SQL interface and schema exploration [34], one for knowledge graphs covering SPARQL and ontology navigation with tool-augmented entity resolution [24], and one for document collections covering retrieval and layout-aware extraction [51]. The execution environment may additionally include higher-level operations such as parameterized query templates or domain-specific extraction routines that extend

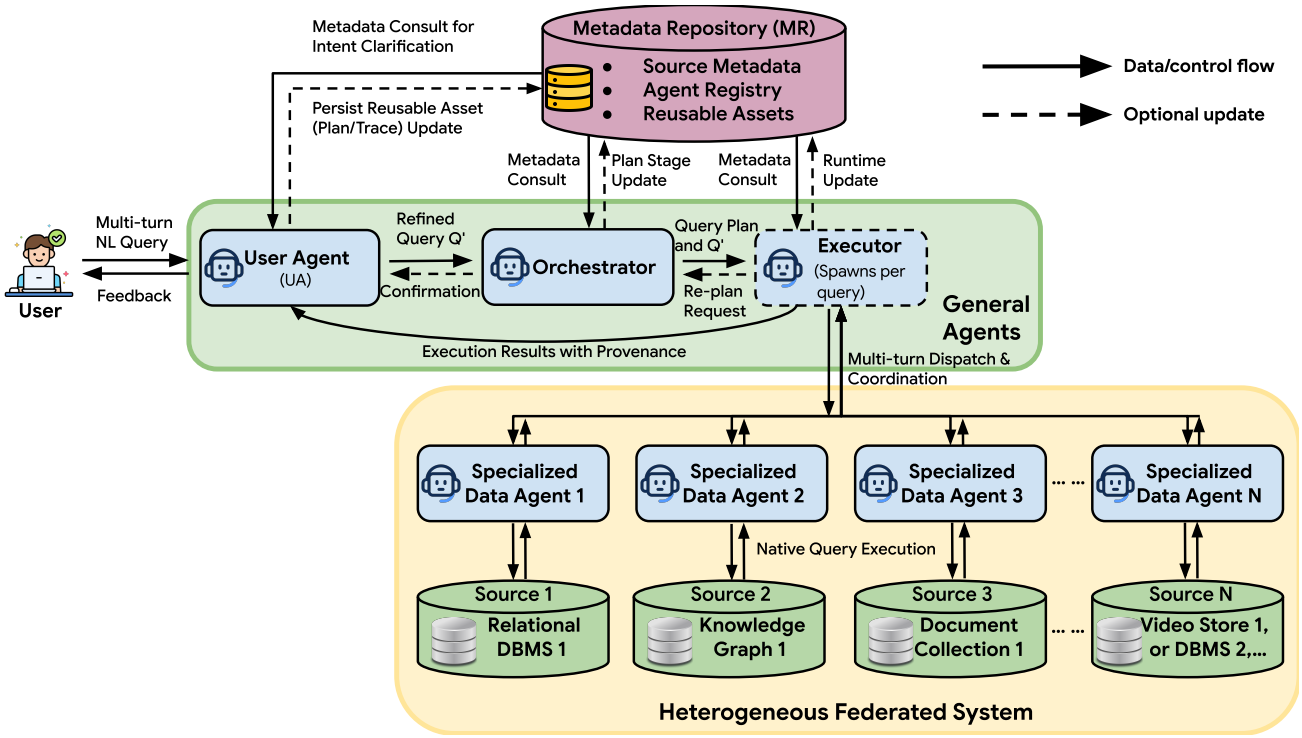


Figure 2: Multi-agent architecture for analytics over heterogeneous federated systems. The User Agent refines queries through multi-turn dialogue grounded in MR; the Orchestrator consults MR to produce an explicit query plan; the ephemeral Executor dispatches subtasks to source-level SDAs via multi-turn coordination, validates intermediate results, and can request re-planning. Each SDA operates within a purpose-built execution environment tailored to its source's native interface. Specialization is at the source level, not the modality level: Source N illustrates that additional sources of any type (e.g., a second relational DBMS or a video store) are each served by their own SDA instance.

the action space beyond raw query execution. Although each agent targets a particular modality, specialization occurs at the source level: two relational DBMSs with different schemas are served by two distinct agent instances.

Metadata Repository (MR). The system maintains a central record of what each source contains and what each agent can do, including source-level metadata, an agent registry, and previously successful query plans. This allows the system to operate without a global schema: it looks up the characteristics of each source in MR when planning a query. When a source is added or changes, only MR is updated. As shown in Figure 2, MR is consulted by all components: the UA for user intent clarification, the Orchestrator for plan-stage metadata lookup, and the Executor for runtime re-routing decisions.

Orchestrator. Given a refined query Q' from the UA (Figure 2), the Orchestrator consults MR and produces a query plan consisting of subtasks assigned to specific sources with explicit dependencies. It does not execute the plan itself but passes the plan and Q' to an Executor. Separating planning from execution prevents any single component from having to both decide what to do and carry it out, which becomes unmanageable as the number of sources and the depth of reasoning grow. The separation also enables parallelism: the Orchestrator can continue planning new queries while multiple Executors concurrently carry out previously issued plans. The explicit plan can be validated before any source is

queried, catching problems such as assigning a subtask to a source whose action space does not include the required operation.

Executor. The Executor carries out the plan by dispatching subtasks to SDAs through multi-turn coordination (Figure 2), managing the flow of intermediate results, and handling parallel and sequential execution. Each Executor is spawned per query and discarded afterward. Before passing any result to the next step, the Executor validates it and can re-route to an alternative source or reformulate the subtask if something goes wrong. When more fundamental issues arise, it sends a re-plan request back to the Orchestrator This intermediate validation is the primary mechanism for preventing per-step errors from compounding through the plan. The Executor's execution trace and successful patterns can be recorded in MR as runtime updates for future reuse.

User Agent (UA). A conversational entry point that receives natural language queries and engages in multi-turn dialogue to clarify ambiguous references, grounded in MR. A reference to "sales data" may correspond to a CRM database or an analytics warehouse; the UA consults MR to determine which sources exist and what they contain, enabling disambiguation grounded in the actual data landscape. On the output side, the UA presents the final answer with provenance information, allowing the user to verify each claim, and can persist successful query plans and execution traces back to MR as reusable assets.

Provenance-by-construction. Every intermediate result carries a record of which source produced it and what evidence supports it. As results are combined across sources, these records are preserved and composed. The final answer includes a complete trace, binding each claim to its source evidence. When sources disagree or evidence is insufficient, the system surfaces the conflict rather than silently choosing one version. We call this provenance-by-construction: claim-evidence bindings are built incrementally as results flow through the pipeline, rather than attached after the fact.

3.4 Open Challenges

This architecture raises several research challenges that the community has not yet addressed.

Query understanding and disambiguation. The UA receives natural language queries that may contain ambiguous references, implicit constraints, or domain-specific terminology. Before any planning can begin, the system must resolve what the user is asking. A reference to "sales data" may correspond to a CRM database, an analytics warehouse, or both; a request for "recent" results requires a specific time range. The UA consults MR to determine which sources exist and what they contain, enabling it to ground the disambiguation in the actual data landscape rather than guessing. When the query remains ambiguous after one pass, the UA engages in multi-turn dialogue to refine the intent before forwarding a resolved query to the Orchestrator. This challenge is a prerequisite for all subsequent steps: an incorrectly interpreted query yields a correct plan for the wrong question.

Dynamic decomposition under unknown source requirements. The Orchestrator must decompose a query into subtasks and assign each to a source, but these decisions are interdependent. The best decomposition depends on which sources are available, and which sources are relevant may only become clear as intermediate results arrive. In the credit risk example, the supplier ownership graph becomes necessary only after the relational aggregation identifies qualifying suppliers. This demands iterative or conditional planning, not just a single upfront decomposition. Existing plan-generation methods do not account for this kind of dynamic source discovery [62].

Routing and execution with intermediate validation. Even with a sound plan, executing it correctly requires matching each subtask to a source whose action space includes the required operation and validating intermediate results before they are consumed downstream. This matching must account for differences both across and within modalities: a subtask requiring graph traversal cannot be sent to a relational DBMS, but equally, a subtask requiring a specific table cannot be sent to an arbitrary DBMS. Without this intermediate validation, per-step errors compound multiplicatively through the plan. Designing validation policies that detect subtle type mismatches or incomplete results without access to ground truth remains an open problem.

Cross-source entity resolution. Entities are represented differently across modalities: a supplier appears as a row with a vendor ID in a relational table, as a named entity in a knowledge graph, and as a name mention in a PDF. Before results from different sources can be correctly combined, these representations must be aligned. This resolution is not a one-time preprocessing step but arises dynamically during execution, because the entities to be aligned depend on intermediate results produced by earlier subtasks.

Cross-modality provenance composition. Database provenance [7] and truth discovery [14] are well-studied individually, but composing provenance across modality boundaries in LLM-based pipelines is largely open. When a claim depends on a reasoning chain that spans all three modalities, the provenance record must compose across these boundaries while remaining interpretable to a human user. The challenge is twofold: reconciling different forms of evidence into a unified trace that a domain expert can follow, and detecting when sources conflict or when the available evidence is insufficient to support a claim.

Source registration and metadata evolution. The architecture depends on MR maintaining an accurate record of each source's action space, metadata, and access constraints. In practice, these records are inherently incomplete even for a static federation, because fully exploring every source upfront is infeasible: a relational DBMS may contain thousands of tables with undocumented columns. The system must therefore operate under partial knowledge and refine its records incrementally as queries reveal new information about what each source contains and can do. Beyond this initial incompleteness, sources also evolve over time: a relational DBMS may add new tables or deprecate old ones, a document collection may be re-indexed with richer metadata, and a knowledge graph may incorporate a large batch of new entities. If MR's records are stale or incomplete, the Orchestrator may route subtasks to sources that cannot perform the required operation, or miss sources that could have contributed. Bootstrapping records for new sources, refining them through query experience, and detecting when existing records have become outdated are all open problems that affect the reliability of planning and routing.

Evaluation gaps. Existing cross-source benchmarks such as HybridQA [10] and MultiModalQA [60] combine a small number of modalities and do not require source discovery, within-modality routing, or provenance assembly. More recent efforts such as FDA-Bench [65] expand modality coverage but do not isolate the contributions of the orchestration layer or evaluate provenance quality. Building benchmarks that jointly test capability-aware routing, provenance assembly, and adaptive re-planning, including gold decomposition plans and gold provenance records, is itself a significant research challenge.

4 Conclusion

In this paper we presented two research projects that are placed on the opposite sides of the LLM-DMS duality. On the DB4LLM side, the project aims to support LLM-agent workloads that go beyond attaching a model to an existing infrastructure; it requires a new design of the data stack. The mismatch between traditional DMS assumptions and the non-deterministic, adaptive behaviour of LLM agents is structural. Therefore, addressing these workloads requires building systems whose features are designed to support agent workloads. On the LLM4DB direction, we describe a project that explores a multi-agent approach for analytical queries over heterogeneous federated systems. In this system specialized data agents, a metadata repository, and a distinct planning-and-execution layer interact to enable complex, multi-step queries expressed in natural language. The queries execute over multiple data sources including relational databases, knowledge graphs and document collections.

Taken together, these two projects illustrate the some of the issues central to the LLM-DMS duality: advances in data management infrastructure make LLM-based analytical systems more reliable and efficient, while the demands of LLM-agent workloads expose limitations in existing DMS designs and motivate new DMS designs. The challenges highlighted by these projects are largely open problems. We hope that framing these problems in terms of the broader duality will encourage the community to engage with both directions together, recognizing that progress on one side creates both demand and opportunity on the other.

References

[1] Kerem Akillioglu. Optimizing data systems for LLM workloads. In *PhD Workshop at VLDB 2025*, 2025. Co-located with VLDB 2025.

[2] Kerem Akillioglu, Anurag Chakraborty, Sairaj Voruganti, and M. Tamer Özsu. Research challenges in relational database management systems for LLM queries. In *Proc. 6th Intl. Wkshp on Applied AI for Database Syst. and Appl.*, 2025. Co-located with VLDB 2025.

[3] Alon Albalak, Yanai Elazar, Sang Michael Xie, Shayne Longpre, Nathan Lambert, Xinyi Wang, Niklas Muennighoff, Bairu Hou, Liangming Pan, Haewon Jeong, Colin Raffel, Shiyu Chang, Tatsunori Hashimoto, and William Yang Wang. A survey on data selection for language models. *Trans. Machine Learning Res.*, 2024. Survey Certification.

[4] Michael R. Anderson, Michael Cafarella, German Ros, and Thomas F. Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *Proc. 35th IEEE Intl. Conf. on Data Engineering*, pages 1466–1477, 2019.

[5] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *Commun. ACM*, 57(8):93–102, August 2014.

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 221–230, 2018.

[7] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Proc. 8th Intl. Conf. on Database Theory*, pages 316–330. Springer Berlin Heidelberg, 2001.

[8] Edward Y. Chang and Longling Geng. Sagallm: Context management, validation, and transaction guarantees for multi-agent LLM planning. *Proc. VLDB Endowment*, 18(12):4874–4886, August 2025.

[9] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: how to use large language models while reducing cost and improving performance. arXiv 2305.05176, 2023.

[10] Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Yang Wang. HybridQA: A dataset of multi-hop question answering over tabular and textual data. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1026–1036, Online, November 2020. Association for Computational Linguistics.

[11] E. F. Codd. Seven steps to rendezvous with the casual user. In J. W. Klimbie and K. L. Koffeman, editors, *Proc. IFIP Working Conf. Data Base Management*, pages 179–200, 1974.

[12] Hanjun Dai, Bethany Y Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya M Phothilimthana, Charles Sutton, and Dale Schuurmans. UQE: A query engine for unstructured databases. *Advances in Neural Information Processing Syst., Proc. 37th Annual Conf. on Neural Information Processing Syst.*, 37:29807–29838, 2024.

[13] Databricks Inc. Apply AI on data using Databricks AI functions. Databricks Documentation, 2025. Last updated Feb 5, 2026. Accessed: 2026-03-01.

[14] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Data fusion: resolving conflicts from multiple sources. In *Proc. 14th Intl. Conf. on Web-Age Information Management*, page 64–76, Berlin, Heidelberg, 2013. Springer-Verlag.

[15] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. Beyond quacking: Deep integration of language models and rag into duckdb. *Proc. VLDB Endow.*, 18(12):5415–5418, August 2025.

[16] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. The BigDAWG polystore system. *ACM SIGMOD Rec.*, 44(2):11–16, 2015.

[17] Raul Castro Fernandez, Aaron J. Elmore, Michael J. Franklin, Sanjay Krishnan, and Chenhao Tan. How large language models will disrupt data management. *Proc. VLDB Endowment*, 16(11):3302–3309, 2023.

[18] Tianxiang Gao and Derrick Li. BigQuery AI supports Gemini 3.0, simplified embedding generation and new similarity function. Google Cloud Blog, January 2026. Accessed: 2026-02-24.

[19] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. arXiv 2312.10997, 2024.

[20] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, December 2001.

[21] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Trans. Database Syst.*, 3(2):105–147, 1978.

[22] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proc. 12th Intl. Conf. on Learning Representations*, 2024.

[23] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL. *IEEE Trans. Knowl. and Data Eng.*, 37(12):7328–7345, December 2025.

[24] Xiangru Jian, Zhengyuan Dong, and M. Tamer Özsu. InteracSPARQL: An interactive system for SPARQL query refinement using natural language explanations. arXiv 2511.02002, 2025.

[25] Saehan Jo and Immanuel Trummer. Thalamusdb: Approximate query processing on multi-modal data. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 2(3), May 2024.

[26] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, December 2012.

[27] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.

[28] Georgia Koutrika. Natural Language Data Interfaces: A Data Access Odyssey. In Graham Cormode and Michael Shekelyan, editors, *Proc. 27TH Intl. Conf. on Database Theory*, volume 290 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[29] Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A. Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, Jun Chen, Markos Markakis, Matthew Russo, Weiyang Wang, Ziniu Wu, Mike Cafarella, Lei Cao, Samuel Madden, and Tim Kraska. KRAMABENCH: A benchmark for AI systems on data-to-insight pipelines over data lakes. In *Proc. 14th Intl. Conf. on Learning Representations*, 2026.

[30] Yunshi Lan, Gaole He, Jinhao Jiang, Jing Jiang, Wayne Xin Zhao, and Ji-Rong Wen. Complex knowledge base question answering: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 35(11):11196–11215, November 2023.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Syst., Proc. 34th Annual Conf. on Neural Information Processing Syst.*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[32] Fei Li and H. V. Jagadish. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endowment*, 8(1):73–84, 2014.

[33] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Syst., Proc. 36th Annual Conf. on Neural Information Processing Syst.*, 2023.

[34] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. OmniSQL: Synthesizing high-quality text-to-sql data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709, July 2025.

[35] Paweł Liskowski, Benjamin Han, Paritosh Aggarwal, Bowei Chen, Boxin Jiang, Nitish Jindal, Zihan Li, Aaron Lin, Kyle Schmaus, Jay Tayade, Weicheng Zhao, Anupam Datta, Nathan Wiegand, and Dimitris Tsirogiannis. Cortex AISQL: A production SQL engine for unstructured data. arXiv 2511.07663, 2025.

[36] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing AI-powered analytics with declarative query processing. In *Proc. 15th Biennial Conf. on Innovative Data Systems Research*, 2025.

[37] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. Optimizing llm queries in relational data analytics workloads. In M. Zaharia, G. Joshi, and Y. Lin, editors, *Proc. 8th Conf. on Machine Learning and Syst.*, volume 7. MLSys, 2025.

[38] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, et al. Supporting our AI overlords: Redesigning data systems to be agent-first. arXiv 2509.00997, 2025.

[39] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*, page 1493–1508, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. Natural language to SQL: State of the art and open problems. *Proc. VLDB Endow.*, 18(12):5466–5471, August 2025.

[41] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Raj Sinha, Jinwoo Shin, and Tomas Pfister. DS-STAR: Data science agent for solving diverse tasks across heterogeneous formats and open-ended queries. arXiv 2509.21825, 2026.

[42] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. Can foundation models wrangle your data? *Proc. VLDB Endow.*, 16(4):738–746, December 2022.

[43] Thomas Neumann and Viktor Leis. A critique of modern sql and a proposal towards a simple and expressive query language. In *Proc. 14th Biennial Conf. on Innovative Data Systems Research*, 2024.

[44] Jingjie Ning, João Coelho, Yibo Kong, Yunfan Long, Bruno Martins, João Magalhães, Jamie Callan, and Chenyan Xiong. Agentic search in the wild: Intents and trajectory dynamics from 14m+ real search requests. arXiv 2601.17617, 2026.

[45] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 4th edition, 2020.

[46] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *VLDB J.*, 33(5):1591–1615, 2024.

[47] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus. *Proc. VLDB Endowment*, 18(11):4171–4184, July 2025.

[48] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc., 2024.

[49] Ralph Peeters, Aaron Steiner, and Christian Bizer. Entity matching using large language models. arXiv 2310.11244, 2024.

[50] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proc. 8th Intl. Conf. on Intelligent User Interfaces*, pages 149–157, New York, NY, USA, 2003. Association for Computing Machinery.

[51] Juan A. Rodriguez, Xiangru Jian, Siba Smarak Panigrahi, Tianyu Zhang, Aarash Feizi, Abhay Puri, Akshay Kalkunte Suresh, François Savard, Ahmed Masry, Shravan Nayak, Rabiul Awal, Mahsa Massoud, Amirhossein Abaskohi, Zichao Li, Suyuchen Wang, Pierre-Andre Noel, Mats Leon Richter, Saverio Vadacchino, Shubham Agarwal, Sanket Biswas, Sara Shanian, Ying Zhang, Sathwik Tejaswi Madhusudhan, Joao Monteiro, Krishnamurthy Dj Dvijotham, Torsten Scholak, Nicolas Chapados, Sepideh Kharaghani, Sean Hughes, M. Özsu, Siva Reddy, Marco Pedersoli, Yoshua Bengio, Christopher Pal, Issam H. Laradji, Spandana Gella, Perouz Taslakian, David Vazquez, and Sai Rajeswar. Bigdocs: An open dataset for training multimodal models on document and code tasks. In *Proc. 13th Intl. Conf. on Learning Representations*, 2025.

[52] Gabriele Sanmartino, Matthias Urban, Paolo Papotti, and Carsten Binnig. The stretto execution engine for LLM-augmented data systems. arXiv 2602.04430, 2026.

[53] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In *Proc. 12th Intl. Conf. on Learning Representations*, 2024.

[54] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *Proc. VLDB Endowment*, 18(9):3035–3048, May 2025.

[55] Junyi Shen, Noppanat Wadlom, and Yao Lu. Batch query processing and optimization for agentic workflows. arXiv 2509.02121, 2026.

[56] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[57] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proc. 41st Intl. Conf. on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR, 23–29 Jul 2023.

[58] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. A survey on employing large language models for text-to-sql tasks. *ACM Comput. Surv.*, 58(2), September 2025.

[59] Michael Stonebraker and Andrew Pavlo. What goes around comes around... and around... *ACM SIGMOD Rec.*, 53(2):21–37, July 2024.

[60] Alon Talmor, Ori Yoran, Amnon Catav, Dan Lahav, Yizhong Wang, Akari Asai, Gabriel Ilharco, Hannaneh Hajishirzi, and Jonathan Berant. MultimodalQA: complex question answering over text, tables and images. In *Proc. 9th Intl. Conf. on Learning Representations*, 2021.

[61] Timescale. pgai. Available at: `https://github.com/timescale/pgai`, 2026. Accessed: 2026-02-25.

[62] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models - A critical investigation. In *Advances in Neural Information Processing Syst., Proc. 36th Annual Conf. on Neural Information Processing Syst.*, 2023.

[63] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. In *Proc. 58th Annual Meeting Assoc. for Computational Linguistics*, pages 7567–7578, 2020.

[64] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? A survey from the language model perspective. In *Proc. 1st Conf. on Language Modeling*, 2024.

[65] Ziting Wang, Shize Zhang, Haitao Yuan, Jinwei Zhu, Shifu Li, Wei Dong, and Gao Cong. FDABench: A benchmark for data agents on analytical queries over heterogeneous data. arXiv 2509.02473, 2025.

[66] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[67] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[68] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *Proc. 1st Conf. on Language Modeling*, 2024.

[69] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. 11th Intl. Conf. on Learning Representations*, 2023.

[70] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: generating structured queries from natural language using reinforcement learning. arXiv 1709.00103, 2017.

[71] Xuanhe Zhou, Xinyang Zhao, and Guoliang Li. LLM-enhanced data management. arXiv 2402.02643, 2024.

Towards AI-Native Data Systems with the Semantic Operator Model and LOTUS

Liana Patel, Carlos Guestrin, Matei Zaharia

Abstract

The semantic capabilities of large language models (LLMs) hold transformative potential for data systems, enabling new language-based analytics over vast knowledge corpora. While LLM-based operations are promising, their integration within traditional data systems is inherently challenging both due to the ambiguity of language-based processing, which makes reasoning about correctness difficult, and the computational expense of scaling LLMs over large datasets. This paper describes semantic operators, a model we have been developing to provide the first formalism for general-purpose AI-based operations with natural language parameters (e.g., filtering, sorting, joining or aggregating records using natural language criteria). Each operator can be implemented by multiple algorithms, which compose individual LLM invocations within a pattern that specifies how to orchestrate the model over the dataset (e.g., a semantic join might be implemented by a nested-loop join algorithm). The semantic operator model defines the expected behavior of each operator with a high-quality reference algorithm, providing a correctness definition for arbitrary LLM-based transformations that go beyond simple batched inference primitives and apply even in the absence of user-labeled data. Our model further provides an optimization framework that reduces execution cost of each operator while providing accuracy statistical guarantees to ensure correct executions. This formalism opens up a rich research and design space towards accurate and efficient LLM-based data processing. In this work we provide an overview of the semantic operator model and extensions to it. Based on our experience deploying semantic operators in the open-source LOTUS system, we also discuss diverse user applications, including agent trace analysis, semantic clustering and deep research systems. Overall, we have been excited to see adoption of the semantic operator model in industry as well as an emerging sub-field of research and believe these early milestones represent steps towards AI-native data systems which will seamlessly integrate semantic reasoning capabilities for rich processing over structured and unstructured data.

1 Introduction

The powerful semantic capabilities of modern language models (LLMs) have created exciting opportunities for building AI-based analytics systems that reason over vast knowledge corpora. Many applications require complex reasoning over vast knowledge corpora. For example a researcher reviewing recent ArXiv [1] preprints may want to quickly obtain a summary of relevant papers from the past week, or find the papers that report the best performance for a particular task and dataset. Similarly, a medical professional may automatically extract biomedical characteristics and candidate diagnoses from many patient reports [2]. Likewise, organizations wish to automatically digest lengthy transcripts from internal meetings and chat histories to validate hypotheses about their business [3].

Each of these tasks require a form of *bulk semantic processing*, where the analytics system must process large amounts of data and orchestrate models in complex patterns across a whole dataset. Supporting the full generality of these applications with efficient, easy-to-use analytics systems would have a transformative impact, similar to what RDBMSes had for tabular data. This prospect, however,

raises two challenging questions: first, *how should developers express semantic queries*, and secondly, *how should we design the underlying data system to achieve high efficiency and accuracy.*

Unfortunately, existing systems are insufficient for bulk semantic processing, either limiting their expressiveness to simple batched inference primitives or providing no accuracy guarantees. First, several systems only support simple batched inference primitives [4–13]. These systems do not support richer LLM-based operations, such as ranking, grouping or joining records, which require more complex *AI algorithms* that compose multiple model invocations to orchestrate the model over the data. Alternatively, more recent LLM-based analytics systems [14, 15] study more complex AI operations, but empirically optimize these operations with *no accuracy guarantees.* These systems lack a formalism to define correct behavior, which hinders their robustness and usability. These obstacles highlight a core challenge of integrating semantic-based processing within reliable query systems due to the inherent ambiguity of natural language instructions and LLM outputs.

Our work propose *semantic operators*, a model we have been developing, which extends the relational model with AI-based operations. Semantic operators provide the first formalism *with statistical accuracy guarantees* for general-purpose AI-based operations with natural language parameters, including semantic filters, joins, top-k rankings, aggregations, and projections. Each operator takes a concise natural language signature, given by the programmer, and its behavior is fully specified by a tractable, high-quality reference algorithm. Our optimization approach then exploits the rich design space of semantic operator execution plans to reduce cost, while providing *statistical accuracy guarantees* for individual operators with respect to the reference algorithm (i.e., ensuring that the output of the optimized operator will be similar to that of the reference algorithm). We have implemented semantic operators in LOTUS (**L**LMs **O**ver **T**ext, **U**nstructured and **S**tructured data), an open source system that exposes these operators in a simple DataFrame-based API.

In this work we provide an overview of the semantic operator model, extensions of the model, and recent applications based on our experience deploying semantic operators in LOTUS. Specifically, we discuss how the semantic operator model, which originally defines the correctness of *individual* semantic operators in the absence of user-labeled data extends to two additional settings: multi-operator queries, and settings with user-labeled data. We discuss optimizations for both of these as well as our recent extensions in LOTUS, including the addition of a lazy API to optimize semantic operator queries for these settings. LOTUS' updated implementation includes logical plan optimizations, such as operator re-ordering, prompt optimization, for accuracy improvments with user-labeled data, and approximation algorithms, for cost-reduction using statistical techniques used in prior works [16, 17] with novel and efficient proxy scores. Lastly, we discuss merging user applications of semantic operators among LOTUS users, highlighting 3 cases studies involving agent trace analysis, semantic clustering [18] and deep research systems for generative research synthesis [19]. We demonstrate how queries for each of these applications are being implemented with LOTUS programs consisting of a few semantic operators. Overall, since open-sourcing LOTUS, we have been excited to see a growing user-base, adoption of semantic operators among large database vendors [20, 21], and a growing body of exciting research work. Ultimately, we believe semantic operators serves as a foundation for the future of AI-native data systems that will serve rich, semantic-based processing over vast knowledge corpora with unprecedented scale, accuracy and efficiency.

2 The Semantic Operator Model

The data independence model of relational systems [22] has had transformative impact on database systems by decoupling application logic from how data is stored and structured. The semantic operator model extends this concept of relational systems and introduces *model-data independence* for AI-based

```
def paper_digest(research_interest: str, baseline: str):
  return papers_df\
    .sem_filter(f"the paper {{abstract}} claims to outperform {baseline}")\
    .sem_agg(f"Write a digest summary based on each {{abstract}} describing how these
    papers relate to {research_interest}")
```

Figure 1: Example LOTUS program using semantic operators to return a summary of relevant papers from the dataset papers_df. The user function, takes two strings, describing a research interest and a research baseline. The program filters papers in the dataset based on whether each paper claims to outperform the baseline. Using the filtered papers the program then constructs a summary.

data processing. Model-data independence decouples application logic from the underlying AI-based algorithm which specifies individual model invocations for processing each data record. The semantic operator model provides this form or independence by defining correct executions and optimizations for *arbitrary* LLM-based transformations over datasets, such as language-based joins, filters, top-k ranking and aggregations. This model provides a declarative data programming abstraction, where programmers write application logic with high-level operators, e.g., `sem_join`, `sem_filter`, `sem_topk`, `sem_agg`—as opposed to low-level LLM() calls—and systems can transparently optimize query execution. In this section, we begin by providing an example of a semantic operator program and will then provide an overview of the semantic operator model, including definitions of semantic operators and correct optimizations as well as common semantic operators.

2.1 Example Semantic Operator Program

To begin to understand the capabilities of semantic operators, we examine a simple program written with the LOTUS API, shown in Figure 1. The function `paper_digest` takes two string parameters, a user's research interest and a research baseline, and returns a digest of relevant research papers that claim to outperform the baseline. The dataset of research papers, `papers_df`, contains an *abstract* attribute. The `paper_digest` function processes this dataset using 2 semantic operators, a semantic filter and a semantic aggregation. The program first performs a semantic filter to find papers with abstracts claiming to outperform the baseline. The program then performs a semantic aggregation to summarize the filtered papers, returning a digest.

Semantic operators perform familiar transformations, such as filters and aggregations, reminiscent of relational operators. The crucial difference between semantic operators and their relational counterparts is that semantic operators are parameterized by *natural-language specifications*. For instance, the `sem_filter` operator takes the natural language predicate "the paper {abstracts} claim to outperform the baseline". The expected output of this operator is all records from the dataset with abstracts that pass the natural language predicate.

Due to each operator's language-based parameters, the behavior of each operator is inherently ambiguous without further specification and will depend on the specific algorithm used. For instance, one naive algorithm to implement a semantic filter might pass the entire dataset to a single LLM call, prompting the model output all rows that pass the user's predicate. A simple but effective alternative might pass *each row* to a single LLM invocation, prompting the model to output a boolean assessing whether the user's predicate holds for that row. The latter option represents a high-accuracy implementation that will avoid issues like long-context degradation [23]. We can use this high-quality algorithm as a reference to define the semantic filters behavior. More generally, each semantic operator has an expected behavior defined by a high-quality *reference algorithm*.

The actual execution of each semantic operator may deviate from the reference algorithm. For example, for the semantic filter, the execution engine might leverage various proxies, such as smaller

Table 1: Summary of Key Semantic Operators. T denotes a relation, X and Y denote arbitrary tuple types. l denotes a parameterized natural language expression ("langex" for short). Each operator may permit additional optional parameters, including accuracy targets.

Operator	Description	Definition	Reference Algorithm
$sem\_filter(l\text{: } X \to Bool)$	Returns the tuples that pass the langex predicate.	$\{t_i \mid t_i \in T \wedge l_M(t_i) = 1\}$	Compute $M(t_i, l), t_i \in T$
$sem\_join(t\text{: } T, \quad l\text{: } (X,Y) \to Bool)$	Joins a table against a second table t by keeping all tuple pairs that pass the langex predicate.	$\{(t_i, t_j) \mid l_M(t_i, t_j) = 1, t_i \in T_1, t_j \in T_2\}$	Compute $M(\{t_i, t_j\}, l), t_i \in T_1, t_j \in T_2$
$sem\_agg(l\text{: } T[X] \to X)$	Aggregates input tuples according to the langex reducer function.	$l_M(t_1, ..., t_n) \forall t_1, .., t_n \in T$	Perform a hierarchical reduce, recursively computing $acc_{i,r} \leftarrow M(\{acc_{n_i,r-1}, ..., acc_{n_i+n,r-1}\}, l)$
$sem\_topk(l\text{: } T[X] \to Seq[X], \quad k\text{: } int)$	Returns an ordered list of the k best tuples according to the langex ranking criteria.	$\langle t_1, ..., t_k \rangle \ st \ \forall (t_i, t_j), i < j \implies l_M(t_i, t_j) = \langle t_i, t_j \rangle$	Perform quick-select top-k using pairwise comparisons, $M(\{t_i, t_j\}, l)$
$sem\_group\_by(l\text{: } X \to Y, \quad C\text{: } int)$	Groups the tuples into C categories based on the langex grouping criteria.	$\underset{\{\mu_1,...\mu_C\}, \mu_i \in V^{\mathbb{N}}}{\text{argmax}} \sum_{t_i \in T} \max_{j \in 1...C} l_M(t_i, \mu_j)$	Obtain centers $\mu_1, ...\mu_C$ with a clustering algorithm, and perform pointwise assignments $M(t_i, (l, \mu_1, ..., \mu_C))$,$t_i \in T$,
$sem\_map(l\text{: } X \to Y)$	Performs the projection specified by the langex.	$\{l_M(t_i) \mid t_i \in T\}$	Compute $M(t_i, l), t_i \in T$

models, embedding based search, keyword search or code synthesis to speed up execution and reserve LLM-based predicate evaluation only when needed. In doing so, the execution engine must still ensure that the resulting filter execution is faithful to the expected filter results, defined by the operator's reference algorithm. Using the semantic operator model, the data management system achieves model-data independence, allowing the programmer to compose their application logic with high-level dataset transformations while guaranteeing that the resulting execution will be close to the expected behavior defined by the model.

2.2 Defining Semantic Operators

Definition. *A semantic operator is a declarative transformation over one or more datasets, parameterized by a natural language expression. Each semantic operator can be implemented by potentially many AI-based algorithms, and its correct behavior is defined with respect to a given reference algorithm.*

Table 1 lists a core set of semantic operators, which cover common semantic transformations in real-world applications and mirror key transformations in relational operators. We have implemented these transformations in LOTUS, and many of them have been recently adopted among existing open-source LLM-based data processing systems [14, 24, 25] and commercial ones [6, 20]. Specific systems may, of course, provide additional semantic operators or utilities beyond the ones we discuss here.

Each semantic operator takes a *parameterized natural language expressions* (*langex* for short), which are natural language expressions that specify a function over one or more attributes. As Figure 1 demonstrates, the langex signature varies for different semantic transformations. For instance, while the `sem_filter` langex signature provides a natural language *predicate*, the `sem_agg` langex is a commutative, associative *aggregator* expression, which here indicates a summarization task over abstracts.

We provide a high-level definition of each semantic transformation with respect to the user langex

```
join_res = papers_df.sem_join(dataset_df, "The paper {abstract:left} uses the {
    dataset_name:right}.")
topk_res = papers_df.sem_topk("The paper has the funniest {title}", K=5)
grouped_res = papers_df.sem_group_by("What is the main research topic of the paper {
    abstract}", C=10)
```

Figure 2: Example usage of sem_join, sem_topk, and sem_groupby. papers_df contains fields for the "title" "abstract", and dataset_df, contains a field called "dataset_name".

and a world model[1], M, which captures a probability distribution over the vocabulary V. For example, semantic filter returns $\{t_i | t_i \in T \wedge l_M(t_i) = 1\}$, where T is the input relation and $l_M(t_i)$ represents a natural language predicate evaluated on tuple t_i with model M. Notably, the definition of each semantic operator can be implemented by multiple AI-based algorithms, and different decisions as to how to invoke the model over the relation have consequences on the algorithm's result quality. Thus, the correct behavior of each semantic operator is specified by a *reference algorithm*, a computable and tractable AI algorithm that produces results considered to be high-quality. Each reference algorithm specifies a model access pattern over the relation T via an algorithm composed of model invocations $M(x, l)$, describing the subset of the data $x \subseteq T$, each model call is invoked over and the task-specific language expressions, given by l.

2.3 Defining Correct Optimizations for Semantic Operators

Semantic operators create a rich design space of diverse execution plans. While reference algorithms provide high-quality implementations, they are often expensive, with the complexity of LLM calls scaling linearly or quadratically in the dataset cardinality. As such, we define correct optimizations for individual semantic operators below by considering alternate AI-based algorithms that can reduce cost while offering *close results*. This model allows an execution system to optimize individual operators even in the absence of user-labeled data. Moreover, this model of can be naturally extended to multi-operator queries and settings with user-labeled data, and we describe opportunities for these optimization regimes in Section 3. In general, optimized semantic operator plans allow for both lossless optimizations and approximations of a reference algorithm. This formulation builds on approximate query processing and assumes some error is often tolerable, which our work finds is often reasonable for achieving high-quality results for semantic processing, which is inherently non-exact.

Definition. *A correct optimization for a given semantic operator, and a reference algorithm for that operator, reduces cost while providing statistical accuracy guarantees with respect to the reference algorithm. Specifically, the optimization should ensure an accuracy target, γ, is met with probability $1 - \delta$.*

2.4 Core Semantic Operators

We now overview several core semantic operators, providing the operator's definition and a reference algorithm for each. We discuss design decisions for our reference algorithms based on state-of-the-art algorithms studied in the AI literature, well-known failure cases, such as long-context challenges [23], or our experimental observations. Our prior work [19, 26, 27] confirms that the reference algorithms we present support state-of-the-art result quality in real ML applications; however, finding optimal reference algorithms for each operator remains an open research question.

Semantic Filter is a unary operator over the relation T and returns the relation $\{t_i | t_i \in T \wedge l_M(t_i) = 1\}$, where the langex provides a natural language predicate over one or more attributes.

Reference Algorithm. Our reference algorithm runs batched LLM calls over all tuples in relation T. Each model invocation, $M(t_i, l)$, prompts the LLM with a single tuple $t_i \in T$, the langex predicate,

[1]In practice, the world model, M may be the strongest LLM a practitioner has available.

and an operator-specific instruction to generate a boolean value. This simple choice avoids well-studied long-context issues [23] by processing rows independently rather than in a single invocation.

Semantic Join provides a binary operator over relations T_1 and T_2 to return the relation $\{(t_i, t_j) | l_M(t_i, t_j) = 1, t_i \in T_1, t_j \in T_2\}$. Here the langex is parameterized by the left and right join keys and describes a natural language predicate over both.

Reference Algorithm. The reference algorithm implements a *nested-loop join pattern*, performing a single predicate evaluation for each pair of tuples, with model invocations $M(\{t_i, t_j\}, l), t_i \in T_1, t_j \in T_2$. This yields an $O(|T_1| \cdot |T_2|)$ LLM call complexity.

Semanic Top-k imposes a ranking[2] over the relation T and returns the ordered sequence, $\langle t_1, ..., t_k \rangle$ st $\forall (t_i, t_j), i < j \implies l_M(t_i, t_j) = \langle t_i, t_j \rangle$. Here, the langex signature is a general ranking criteria.

Reference Algorithm. Two important algorithmic design decisions for the semantic top-k include how to implement LLM-based comparison and how to aggregate ranking information from these comparisons. Our reference algorithm uses pairwise LLM comparisons for the former, and a quick-select top-k algorithm [28] for the latter. We briefly describe the reason for these choices and alternatives considered. Our decisions build on prior works, which have studied LLM-based passage re-ranking [29–37] and ranking with noisy comparisons [38, 39] with the goal of achieving high quality results in a modest complexity of LLM calls or comparisons.

First, pairwise-prompting methods offer a simple and high-quality approach that feeds a single pair of tuples to each LLM invocation, $M(\{t_i, t_j\}, l)$, prompting the model to compare the two inputs and output a binary label. The two main classes of alternatives are point-wise ranking methods [29–31, 36, 40], and list-wise ranking methods [32–34, 37], both of which have been shown to face quality issues [29, 35, 37]. Our prior work verifies these limitations [27]. In contrast, pairwise comparisons have been shown to be effective and relatively robust to input ordering [35].

In addition, we consider several possible rank-aggregation algorithms, including quadratic sorting algorithms, a heap-based top-k algorithm and a quick-select-based top-k ranking algorithm. Our prior work [27] demonstrates that each of these sorting algorithms yield high-quality results, comparable to one another. However, the quick-select-based algorithm offers an efficient implementation with at least an order of magnitude fewer LLM calls than the quadratic sorting algorithm and more opportunities for efficient batched inference, leading to lower execution time, compared to a heap-based implementation. The quick-select top-k algorithm proceeds in successive rounds, each time choosing a pivot, and comparing all other remaining tuples to the pivot tuple to determine the rank of the pivot. Because each round is fully parallelizable, we can efficiently batch these LLM-based comparisons before recursing.

Semantic Aggregation performs a many-to-one reduce over the input relation, returning $l_M(t_1, ..., t_n)$, $\forall t_1, .., t_n \in T$. Here, the langex signature is a commutative, associative aggregation function[3], which can be applied over any subset of rows to produce an intermediate results. We note that the langex iteself is model-agnostic, assuming infinite context. Managing finite context limits of the underlying model M is an implementation detail of the system.

Reference Algorithm. Our reference algorithm uses a hierarchical reduce pattern. Our choice builds on the LLM-based summarization pattern studied by prior research works [41–43] and deployed systems [44, 45], which we briefly overview. Prior works primarily study two aggregation patterns. First, a fold pattern performs a linear pass over the data, iteratively updating the accumulated partial answer with the next tuple t_i, given by $acc_i \leftarrow M(\{acc_{i-1}, t_i\}, l)$. Alternatively, the hierarchical reduce pattern recursively aggregates n inputs in each round r and produce multiple partial answers, given by $acc_{i,r} \leftarrow M(\{acc_{n_i, r-1}, ..., acc_{n_i+n, r-1}\}, l)$ until a single answer remains. Both represent candidate

[2]This definition implies that l_M imposes a total and consistent ordering. However, this definition can also be softened to assume partial orderings and noisy comparisons with respect to model M.

[3]We note that the ordering of inputs within an LLM prompt invocation can in fact affect results quality for some tasks. To allow programmers to override the commutativity and associativity, LOTUS exposes a partioner function.

reference algorithms, however, the hierarchical pattern has been shown to produce higher quality results for commutative, associative aggregation tasks, like summarization, in prior work [42] and allows for greater parallelism during query processing, making it our default choice.

Semantic Group-by takes a langex that specifies a projection from a tuple to an unknown group label, as well as a target number of groups, which specifies the desired granularity of group labels. As a example, a user might group-by the topics presented in a set of ArXiv papers, wishing to find 10 key groups. The group-by operator must *discover* representative group labels and assign a label to each each tuple. In general, performing the unsupervised group discovery is a clustering task, which is NP-hard [46]. Clustering algorithms over points in a metric space typically optimize the potential function tractably using coordinate descent algorithms, such as k-means. For the semantic group-by, the clustering task is over unstructured fields with a natural language similarity function specified by $l_M(t_i, \mu_j)$, which imposes a real-valued score between a tuple t_i and a candidate label μ_j. This operator poses the following optimization problem:

$$\underset{\{\mu_1,...\mu_C\},\ \mu_i \in V^{\mathbb{N}}}{\operatorname{argmax}} \sum_{t_i \in T} \max_{j \in 1...C} l_M(t_i, \mu_j)$$

where μ_i is a group labels, consisting of tokens in vocabulary, V.

Reference Algorithm. Since this operator, by definition, entails the NP-hard clustering problem [46], our reference algorithm uses a tractable clustering heuristic to discover group labels, then performs point-wise classification to assign each record to a discovered group label. Specifically, our LLM-based clustering algorithm discovers centers $\mu_1, ..., \mu_C$ by first performing a semantic projection, with model invocations $M(t_i, l), t_i \in T$, prompting the LLM to predict a candidate label for each input tuple. Then, we embed these candidate labels and perform an efficient vector clustering using k-means to construct C groups. For each group, we top-k sample by centroid-similarity scores, and perform a semantic aggregation to synthesize an appropriate label over each group. This provides a reasonable clustering heuristic similar to prior work [15], although alternative heuristics are possible. In the second stage, our reference algorithm uses the C generated labels, $\mu_1, ...\mu_C$, and performs point-wise assignments $M(t_i, (l, \mu_1, ..., \mu_C)), t_i \in T$. We choose point-wise classification to avoid the long-context scaling challenges studied in prior works [15, 23]. This algorithm yields $O(|T|)$ LLM call complexity.

3 Optimized Execution Plans for Semantic Operators

Semantic operators open up a rich design space for optimizations to reduce cost while providing statistical accuracy guarantees. This design space includes novel opportunities specific to the unique properties of LLM-based execution, as well as the application of traditional optimizations from relational data processing (e.g. operator re-ordering [11–13, 47]). In this section, we begin by describing our existing research studying optimizations for semantic operator queries. We overview three optimization regimes of the semantic operator model. We will begin with optimizations for individual operators (3.1), which represents the simplest setting and follows directly from the definitions provided in Section 2 involving cost reduction for individual operators with respect to their reference algorithm. We then discuss extensions of the semantic operator model to optimizations for multi-operator queries (Section 3.2), and to optimizations with user-labeled data (Section 3.3), which open up additional opportunities.

3.1 Optimizations for Individual Operators

To begin, we consider optimizations for the relatively simple setting that involves a single semantic operator and no user-labeled data. Here, we directly apply the definition of a correct optimization from Section 2.3, and our goal is to reduce the execution cost of the semantic operator while providing

a statistical accuracy guarantee for the execution with respect to the operator's reference algorithm. Excitingly, even this seemingly simple setting of single-operator optimizations already opens up a tremendous design space that can be transparently exploited by system optimizers, creating new research opportunities. In fact, in our prior work [27] we demonstrate up to 1000× speedups with accuracy guarantees with respect to an operator's reference algorithm.

Two key mechanisms for reducing cost of each operator are the use of proxies and cost-based planning. First, proxies refer to any means of approximating the LLM-based invocations of the reference algorithm. This could be a small language model, code-based execution, an embedding-similarity score, or other tool executions. The use of proxies in conjunction with statistical techniques from approximate query processing often allows for significant cost reduction with statistical accuracy guarantees. Secondly, cost-based planning allows the optimizer to enumerate multiple possible AI algorithms—each of which may use one or more proxies—before choosing to execute the lowest cost one to exploit the accuracy-cost tradeoff space. This is often necessary since the effectiveness of a candidate plan with particular models and proxies may vary widely depending on the specific task and dataset given by the user's query.

3.1.1 Example: Semantic Joins

To provide an example, we consider the semantic join operator, where the operator's accuracy metrics are precision and recall. The reference algorithm for the semantic join invokes an LLM over every pair of tuples from the right and left join table, prompting the model to output a boolean value indicating whether the predicate holds for the input tuple pair. This reference algorithm has a quadratic LLM call complexity with respect to the cardinality of the input tables, making it expensive at the size of datasets scale. One way we can approximate this algorithm is using a proxy model to perform predicate evaluations in place of the LLM-based predicate evaluation when possible—this represents a *proxy-join* pattern. In our prior work [27], we experiment with embedding-based proxies that provide similarity scores between the right and left join key of each tuple pair. Our method uses model cascades[48, 49, 49–52] to decide when to use the proxy model or resort to the LLM based on the accuracy guarantees requested by the user query. When the user predicate is easy to approximate with embedding similarity, we expect this approximation to reduce cost significantly. For instance, if the user predicate is "the {article:left} is relevant to the {topic:right}", we might expect many LLM calls to be well-approximated similarity scores taken between each article embedding and each topic embedding. Since the proxy is far cheaper than an LLM call, using this approximation can significantly reduce cost while maintaining high accuracy.

However, in other cases, embedding similarities may be too weak of a proxy. For instance, suppose the user predicate is "the {article:left} claims to outperform the method_name:right}", where the right table is a dataset about research methods with an attribute for method_name, and the left table contains research papers and an attribute for the article. Here, the predicate requires more nuanced reasoning that embedding similarity scores between the article text and method name cannot sufficiently capture. As a result, an alternative algorithm, a *project-proxy-join* pattern, might be more appropriate, where the system first projects the right or left join keys to make their domains more similar, allowing the proxy, in this case embedding similarities, to perform better. A simple form of projections which we have studied [27] provides the LLM with the left join key and prompts it to predict the right join key based on the predicate relationship. In this example, the algorithm would invoke an LLM over each article, prompting the model to extract the method names which the article claims to outperform. Similar to the proxy-join pattern, this algorithm then uses model cascades to leverage the proxy when possible while ensuring statistical accuracy guarantees. These two join patterns represent just two candidate plans, but many others could be generated by a query optimizer, and finding new join plans remains an open research question with exciting recent work emerging in this area [53]. Crucially, once multiple

plans are generated, the optimizer must pick the lowest-cost plan to optimize for the given user query, configured models, and dataset. Efficient search over many possible plans, with possibly many choices of proxies and alternative algorithms remains an interesting, open area of research.

3.2 Extension to Multi-Operator Queries

We now consider an extension of the semantic operator model to multi-operator queries, containing one or more semantic operators possibly composed with relational operators. In this setting, the user specifies accuracy targets at the *query-level*, and query's reference algorithm is constructed by composing together the reference algorithm of individual semantic operators. We have been actively exploring optimizations in this setting, and recent research likewise studies this space, demonstrating strong promise for achieving global-level accuracy guarantees by directly extending the semantic operator model [54]. In the future, we envision a rich space of logical optimizations, many of which may apply techniques from relational query processing, such as operator re-ordering or fusion. The optimizer is then responsible for assigning the appropriate error budget to each logical semantic operator to reduce cost and meet the accuracy guarantees of the composite query. For instance, if a user composes a query with two semantic filters, the optimizer might fuse these filters into a single logical filter, assigning the query-level error budget entirely to the fused filter. Alternatively, the optimizer may retain the two logical filters as separate operators, and assign a higher error budget to the filter with the more difficult predicate evaluation, requiring a search and cost-based plan selection to jointly optimize the error-budgets assigned to both individual operators.

3.3 Optimizations with Labeled Data

So far we have considered the general setting, where the user provides only a semantic operator query, and the optimizer must explore, select and execute plans in the absence of any ground truth data. In this case, the optimizer's goal is cost reduction with accuracy guarantees with respect to the reference algorithm of the provided query. However, if the user provides a labeling function to score the correctness of query results, a wider scope of optimizations are possible, including changes to the reference algorithm itself. Each reference algorithm, as given by a user-written semantic operator query, is specified by user-programmed natural language expression, a configured model and the operators' transformations. These parameters can be optimized against the user-provided labeling function creating a large space of optimization over prompts [55, 56], model selection, and logical plans. Since our initial work implementing semantic operators in LOTUS [26], we have extended the system to support this setting with prompt optimizations and logical plan updates, such as operator re-ordering, for multi-operator queries. Tractably searching the accuracy-cost tradeoff space and ensuring reliability of selected plans remains an open research question that holds promise for future work.

4 Application Case Studies

Since our initial work introducing semantic operators and our open-source implementation in LOTUS [26], we have been excited to see a diverse set of user applications and emerging workloads that underscore the need for these new primitives for semantic bulk processing. In this section, we share our experience deploying semantic operators in the LOTUS open-source system and highlight several exciting case studies, including agent trace analysis, semantic clustering and generative research synthesis.

```
# find examples of agent failures
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")

# explain each failure
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
              .sem_map("summarize the failure cause of the agent {trace}")

# summarize common failure patterns
traces_dataset.sem_filter("the {trace} demonstrates the agent failed the task")\
              .sem_map("summarize the key failure and cause from the agent {trace}",
suffix="agent_failure_summary")\
              .sem_agg("given each {agent_failure_summary}, summarize recurring
failure patterns and way to improve my agent")
```

Figure 3: Example LOTUS program using semantic operators for agent trace analysis. The dataset, traces_dataset, is a large dataset of agent traces with the attribute, trace, containing the long-form text of each agent trace.

4.1 Agent Trace Analysis

As agentic systems become increasingly more pervasive, agent traces serve as rich artifacts capturing common agent patterns, failure modes and unexpected behaviors. As a result, effective tools for processing these large unstructured datasets hold potential to unlock new insights critical for improving deployed agents. LOTUS users have used semantic operator queries for this purpose, automating insights which would otherwise require tedious manual inspection. Typically, agent traces can be long, containing thousands of tokens per trace, making manual inspection over even dozens to hundreds of execution traces difficult and time-consuming. Figure 3 provides several example user queries. The first one demonstrates a simple semantic filter, where the user processes the dataset, `traces_dataset` with the text attribute `trace`, to find traces that demonstrate a failure. The next query chains on a semantic projection, taking the traces with agent failures and summarizing the key cause of the failure in each one. In the final query, the user program chains a final semantic aggregation, which takes all failure summaries of individual traces from the `agent_failure_summary` attribute and creates a single summary of recurring patterns and ways the user could improve the agent.

4.2 Semantic Clustering and Classification

Another common use case among LOTUS users is semantic clustering to taxonomize and classify records in an unstructured dataset. In this application, users have large unstructured datasets, and they must first *discover* key groups before classifying each record in the dataset. One example of this comes from recent research with collaborators studying agents in production [18]. The study involved processing large datasets of human-written responses provided as free-form text and extracting insights. For example, in one question, users listed applications and use cases they were using agents for, and the survey analysis aimed to taxonomize these responses according to 5-10 key domains and classify each response using these labels. As shown in Figure 4, a semantic aggregation is used to first surface patterns across the human-written survey answers to identify candidate labels to produce a taxonomy. In the study, researchers then validated these labels and chose a subset as the taxonomy labels (e.g., Technology, Finance & Banking, Corporate Services, Legal & Compliance, etc). These labels were then used to classify each individual human response, which as the figure shows, can be performed with a semantic projection. We have also seen similar patterns in other settings, including research analysis that processes LLM responses [57]. In these settings of exploratory data analysis, we observe iterative development with a human in the loop is often a key component, for which we find LOTUS' API with an option for eager execution useful.

```
# taxonomize survey response data and list candidate labels
responses.sem_agg("list the key domains for agent use based on the survey responses
given by {Q4_answers}")

# label each response using the produced taxonomy
responses.sem_map(f"classify the {{Q4_answers}} according to one of the following
labels: {taxonomy_labels}")
```

Figure 4: Example LOTUS program using semantic aggregation to surface candidate taxonomy labels, followed by classification with a semantic map using chosen the labels, taxonomy_labels.

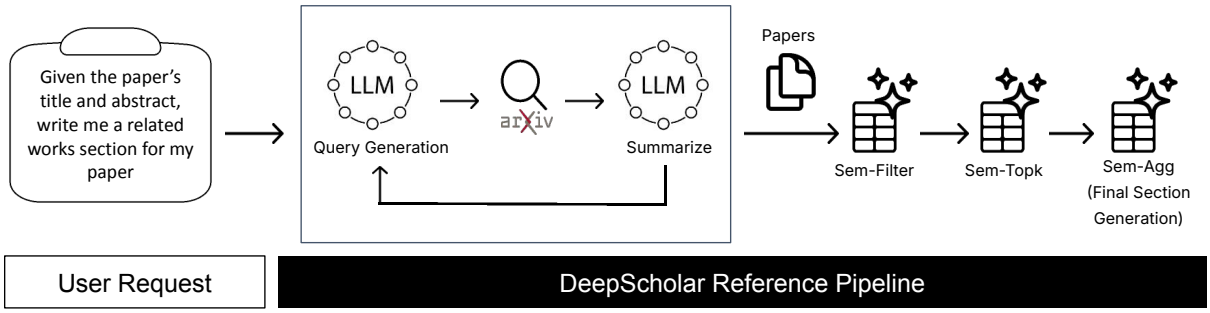


Figure 5: Overview of DeepScholar-ref. The system iteratively writes queries and performs web search, before passing the search results through series of semantic operators using the LOTUS system for LLM-based data-processing, including filtering step to discard irrelevant sources, a top-k ranking step to find most relevant sources, and an aggregation step to generate the final report from all remaining sources.

4.3 Deep Research and Generative Research Synthesis

Recently, systems for *generative research synthesis* have emerged, promising to automate synthesis tasks that require processing large numbers of sources to produce long-form reports. These synthesis tasks are complex and traditionally demand hours of literature searching, reading and writing by human experts. Over the past few months, we have deployed DeepScholar, a deep research system for generative research synthesis which has over 12.8 thousand research queries and thousands of users. DeepScholar is a system implemented on top of LOTUS using our open-source pipeline, DeepScholar-ref, for semantic bulk processing. As Figure 5 shows, DeepScholar-ref decomposes the difficult research synthesis task into semantic operator query. After iteratively searching for documents, the retrieved source set is processed with a semantic filtering step, which assesses whether each document is relevant to the user query. Our pipeline optionally performs a semantic ranking to retain the most relevant documents and then performs a semantic aggregation to produce the final report which answers the user query.

We measure the performance of our open-source reference pipeline using Deepscholar-bench [19], which we developed as a systematic benchmark for generative research synthesis to provide an automated framework and holistically measure performance of systems across three key dimensions—knowledge synthesis, retrieval quality, and verifiability. We find that DeepScholar-ref, which offers competitive performance to OpenAI's Deepresearch while being 4.3× cheaper and 2.28× faster. Notably, even with the same or weaker models, DeepScholar-ref attains up to 6.3× higher Verifiablility scores, which likely reflects the effectiveness of semantic operators as primitives for task decomposition and synthesis over large datasets.

5 Conclusion

In this paper, we provided an overview of our ongoing research developing the semantic operator model, the first formalism for general-purpose, AI-based operations using natural language parameters. Our work introduces a new set of expressive, language-based operators—including filters, joins, top-k ranking and aggregations—and provides a definition for correct operator executions and optimizations. This model naturally extends to multi-operator queries and settings with user-labeled data, creating rich design spaces for optimization that open up exciting areas for new research. The diverse applications we have seen among LOTUS users underscores the value semantic operators hold as key primitives for LLM-based data analysis and semantic processing over large datasets. Since open-sourcing LOTUS, we have seen a growing user-base, as well as adoption of semantic operators among large database vendors, and a growing body of exciting research work. Ultimately, we believe semantic operators serves as a foundational piece towards the future of AI-native data systems that will enable rich semantic-based processing over vast knowledge corpora.

References

[1] "arXiv.org ePrint archive."

[2] K. D'Oosterlinck, F. Remy, J. Deleu, T. Demeester, C. Develder, K. Zaporojets, A. Ghodsi, S. Ellershaw, J. Collins, and C. Potts, "BioDEX: Large-Scale Biomedical Adverse Drug Event Extraction for Real-World Pharmacovigilance," Oct. 2023. arXiv:2305.13395 [cs].

[3] "Discovery Insight Platform."

[4] MotherDuck, "Introducing the prompt() Function: Use the Power of LLMs with SQL! - MotherDuck Blog."

[5] WilliamDAssafMSFT, "Intelligent Applications - Azure SQL Database," Dec. 2024.

[6] "Large Language Model (LLM) Functions (Snowflake Cortex) | Snowflake Documentation."

[7] "LLM with Vertex AI only using SQL queries in BigQuery."

[8] "AI Functions on Databricks."

[9] "Large Language Models for sentiment analysis with Amazon Redshift ML (Preview) | AWS Big Data Blog," Nov. 2023. Section: Amazon Redshift.

[10] S. Liu, A. Biswal, A. Cheng, X. Mo, S. Cao, J. E. Gonzalez, I. Stoica, and M. Zaharia, "Optimizing LLM Queries in Relational Workloads," Mar. 2024. arXiv:2403.05821 [cs].

[11] S. Liu, J. Xu, W. Tjangnaka, S. J. Semnani, C. J. Yu, and M. S. Lam, "SUQL: Conversational Search over Structured and Unstructured Data with Large Language Models," Mar. 2024. arXiv:2311.09818 [cs].

[12] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, and G. Vitagliano, "A Declarative System for Optimizing AI Workloads," May 2024. arXiv:2405.14696 [cs].

[13] Y. Lin, M. Hulsebos, R. Ma, S. Shankar, S. Zeigham, A. G. Parameswaran, and E. Wu, "Towards Accurate and Efficient Document Analytics with Large Language Models," May 2024. arXiv:2405.04674 [cs].

[14] S. Shankar, T. Chambers, T. Shah, A. G. Parameswaran, and E. Wu, "DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing," Dec. 2024. arXiv:2410.12189 [cs].

[15] H. Dai, B. Y. Wang, X. Wan, B. Dai, S. Yang, A. Nova, P. Yin, P. M. Phothilimthana, C. Sutton, and D. Schuurmans, "UQE: A Query Engine for Unstructured Databases," Nov. 2024. arXiv:2407.09522 [cs].

[16] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, Y. Sun, and M. Zaharia, "Accelerating approximate aggregation queries with expensive predicates," *Proceedings of the VLDB Endowment*, vol. 14, pp. 2341–2354, July 2021.

[17] D. Kang, E. Gan, P. Bailis, T. Hashimoto, and M. Zaharia, "Approximate selection with guarantees using proxies," *Proceedings of the VLDB Endowment*, vol. 13, pp. 1990–2003, Aug. 2020.

[18] M. Z. Pan, N. Arabzadeh, R. Cogo, Y. Zhu, A. Xiong, L. A. Agrawal, H. Mao, E. Shen, S. Pallerla, L. Patel, S. Liu, T. Shi, X. Liu, J. Q. Davis, E. Lacavalla, A. Basile, S. Yang, P. Castro, D. Kang, J. E. Gonzalez, K. Sen, D. Song, I. Stoica, M. Zaharia, and M. Ellis, "Measuring Agents in Production," Feb. 2026. arXiv:2512.04123 [cs].

[19] L. Patel, N. Arabzadeh, H. Gupta, A. Sundar, I. Stoica, M. Zaharia, and C. Guestrin, "DeepScholar-Bench: A Live Benchmark and Automated Evaluation for Generative Research Synthesis," Feb. 2026. arXiv:2508.20033 [cs].

[20] "python-bigquery-dataframes/notebooks/experimental/semantic_operators.ipynb at main · googleapis/python-bigquery-dataframes."

[21] P. Liskowski, B. Han, P. Aggarwal, B. Chen, B. Jiang, N. Jindal, Z. Li, A. Lin, K. Schmaus, J. Tayade, W. Zhao, A. Datta, N. Wiegand, and D. Tsirogiannis, "Cortex AISQL: A Production SQL Engine for Unstructured Data," Nov. 2025. arXiv:2511.07663 [cs].

[22] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," vol. 13, no. 6, 1970.

[23] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," Nov. 2023. arXiv:2307.03172 [cs].

[24] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, and G. Vitagliano, "Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing," 2025.

[25] S. Jo and I. Trummer, "ThalamusDB: Approximate Query Processing on Multi-Modal Data," *Proc. ACM Manag. Data*, vol. 2, pp. 186:1–186:26, May 2024.

[26] L. Patel, S. Jha, C. Guestrin, and M. Zaharia, "LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data," July 2024. arXiv:2407.11418 [cs] version: 1.

[27] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia, "Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS," *Proceedings of the VLDB Endowment*, vol. 18, pp. 4171–4184, July 2025.

[28] C. A. R. Hoare, "Algorithm 65: find," *Commun. ACM*, vol. 4, pp. 321–322, July 1961.

[29] S. Desai and G. Durrett, "Calibration of Pre-trained Transformers," Mar. 2020.

[30] A. Drozdov, H. Zhuang, Z. Dai, Z. Qin, R. Rahimi, X. Wang, D. Alon, M. Iyyer, A. McCallum, D. Metzler, and K. Hui, "PaRaDe: Passage Ranking using Demonstrations with Large Language Models," Oct. 2023. arXiv:2310.14408 [cs].

[31] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove, C. D. Manning, C. Ré, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. Wang, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda, "Holistic Evaluation of Language Models," Nov. 2022.

[32] X. Ma, X. Zhang, R. Pradeep, and J. Lin, "Zero-Shot Listwise Document Reranking with a Large Language Model," May 2023.

[33] R. Pradeep, S. Sharifymoghaddam, and J. Lin, "RankVicuna: Zero-Shot Listwise Document Reranking with Open-Source Large Language Models," Sept. 2023. arXiv:2309.15088 [cs].

[34] R. Pradeep, S. Sharifymoghaddam, and J. Lin, "RankZephyr: Effective and Robust Zero-Shot Listwise Reranking is a Breeze!," Dec. 2023. arXiv:2312.02724 [cs].

[35] Z. Qin, R. Jagerman, K. Hui, H. Zhuang, J. Wu, L. Yan, J. Shen, T. Liu, J. Liu, D. Metzler, X. Wang, and M. Bendersky, "Large Language Models are Effective Text Rankers with Pairwise Ranking Prompting," Mar. 2024. arXiv:2306.17563 [cs].

[36] D. Sachan, M. Lewis, M. Joshi, A. Aghajanyan, W.-t. Yih, J. Pineau, and L. Zettlemoyer, "Improving Passage Retrieval with Zero-Shot Question Generation," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, (Abu Dhabi, United Arab Emirates), pp. 3781–3797, Association for Computational Linguistics, 2022.

[37] W. Sun, L. Yan, X. Ma, S. Wang, P. Ren, Z. Chen, D. Yin, and Z. Ren, "Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents," Apr. 2023.

[38] N. B. Shah and M. J. Wainwright, "Simple, Robust and Optimal Ranking from Pairwise Comparisons," Apr. 2016. arXiv:1512.08949 [cs, math, stat].

[39] M. Braverman and E. Mossel, "Noisy sorting without resampling,"

[40] S. Wu, S. Zhao, M. Yasunaga, K. Huang, K. Cao, Q. Huang, V. N. Ioannidis, K. Subbian, J. Zou, and J. Leskovec, "STaRK: Benchmarking LLM Retrieval on Textual and Relational Knowledge Bases," Apr. 2024.

[41] J. Wu, L. Ouyang, D. M. Ziegler, N. Stiennon, R. Lowe, J. Leike, and P. Christiano, "Recursively Summarizing Books with Human Feedback," Sept. 2021. arXiv:2109.10862 [cs].

[42] Y. Chang, K. Lo, T. Goyal, and M. Iyyer, "BooookScore: A systematic exploration of book-length summarization in the era of LLMs," Apr. 2024. arXiv:2310.00785 [cs].

[43] G. Adams, A. Fabbri, F. Ladhak, E. Lehman, and N. Elhadad, "From Sparse to Dense: GPT-4 Summarization with Chain of Density Prompting," Sept. 2023. arXiv:2309.04269 [cs].

[44] "Querying - LlamaIndex 0.9.11.post1."

[45] "LangChain."

[46] S. Dasgupta, "The hardness of k-means clustering,"

[47] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating Machine Learning Inference with Probabilistic Predicates," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), pp. 1493–1508, Association for Computing Machinery, May 2018.

[48] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, (Kauai, HI, USA), pp. I–511–I–518, IEEE Comput. Soc, 2001.

[49] M. Yue, J. Zhao, M. Zhang, L. Du, and Z. Yao, "Large Language Model Cascades with Mixture of Thoughts Representations for Cost-efficient Reasoning," Feb. 2024. arXiv:2310.03094 [cs].

[50] L. Chen, M. Zaharia, and J. Zou, "FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance," May 2023. arXiv:2305.05176 [cs].

[51] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "NoScope: Optimizing Neural Network Queries over Video at Scale," Aug. 2017. arXiv:1703.02529 [cs].

[52] D. Kang, E. Gan, P. Bailis, T. Hashimoto, and M. Zaharia, "Approximate Selection with Guarantees using Proxies," Jan. 2022. arXiv:2004.00827 [cs].

[53] I. Trummer, "Implementing Semantic Join Operators Efficiently," Oct. 2025. arXiv:2510.08489 [cs].

[54] G. Sanmartino, M. Urban, P. Papotti, and C. Binnig, "The Stretto Execution Engine for LLM-Augmented Data Systems," Feb. 2026. arXiv:2602.04430 [cs].

[55] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou, "TextGrad: Automatic "Differentiation" via Text," June 2024. arXiv:2406.07496 [cs].

[56] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines," Oct. 2023.

[57] L. Dunlap, K. Mandal, T. Darrell, J. Steinhardt, and J. E. Gonzalez, "VibeCheck: Discover and Quantify Qualitative Differences in Large Language Models," Apr. 2025. arXiv:2410.12851 [cs].

Optimal Block Nested Loops Implementations for Semantic Joins

Immanuel Trummer
Cornell University
itrummer@cornell.edu

Abstract

Semantic query processing engines often support semantic joins, enabling users to match rows that satisfy conditions specified in natural language. Such join conditions can be evaluated using large language models (LLMs) that solve novel tasks without task-specific training.

Currently, many semantic query processing engines implement semantic joins via nested loops, invoking the LLM to evaluate the join condition on row pairs. Instead, this paper proposes a novel algorithm, inspired by the block nested loops join operator implementation in traditional database systems. The proposed algorithm integrates batches of rows from both input tables into a single prompt. The goal of the LLM invocation is to identify all matching row pairs in the current input. The paper introduces formulas that can be used to optimize the size of the row batches, taking into account constraints on the size of the LLM context window (limiting both input and output size). A formal analysis of asymptotic processing costs, as well as empirical results, demonstrates that the proposed approach reduces costs significantly and performs well compared to join implementations used by recent semantic query processing engines.

1 Introduction

Several recent systems [2, 4, 11, 18, 23, 26–28] expand SQL by introducing semantic operators. Those operators, including, for instance, semantic filters and semantic sort operators, are configured via natural language instructions and evaluated by large language models (LLMs). Compared to traditional relational operators, the per-byte processing overheads of such operators are typically higher by many orders of magnitude. This means, in the context of semantic queries, processing overheads are typically dominated by overheads due to semantic operators. This makes it crucial to make those operators as efficient as possible.

This paper focuses on a semantic version of a classical relational operator: the relational join. Semantic joins, as defined by systems like LOTUS [22], enable users to describe the join condition in natural language. Note that this paper does not explicitly focus on equality joins. Instead, it focuses on general theta-joins [17] with natural language predicates. For instance, such join operators are useful in the following example scenario.

Example 1: To investigate a large corporation, prosecutors plan to analyze a large collection of emails. The goal is to compare emails to statements made by executives of that company. Of particular interest are instances where an email contradicts statements made by the defendants. This can be modeled as a join between two data sets, one containing statements and the other one containing emails. The join predicate can be formulated in natural language as "pairs of documents that contradict each other."

A naive implementation of the semantic join operator, implemented in several semantic query processing engines at the time of writing, uses the LLM to evaluate the join condition on each pair of input tuples (i.e., a tuple nested loops join). The amount of data sent to the LLM is directly proportional to the product of the cardinalities of both input tables. Similarly, the amount of data generated by the LLM (one token per pair of input tuples with the predicate evaluation result) is proportional to the cardinality product. LLM providers such as OpenAI and Anthropic calculate processing fees as a weighted sum over the number of input and output tokens. Hence, monetary processing fees are proportional to the cardinality product as well.

The aforementioned approach uses the LLM like a user-defined function predicate. However, this approach does not exploit the flexibility of LLMs, enabling them to follow almost arbitrary instructions in natural language. Instead of merely returning a Boolean result for each tuple pair, we can instruct the LLM to find all matching pairs, according to the join predicate, when given not two tuples but two larger batches of tuples in the input.

We can exploit the flexibility of LLMs for a block nested loops implementation of the semantic join operator. In each call to the LLM, we send tuple batches from both input tables and retrieve all matching pairs in the LLM reply. Clearly, this approach reduces the number of LLM calls, as the number of pairs of tuple batches is smaller than the number of tuple pairs (assuming that batches contain at least two tuples). However, the amount of data read and generated in each LLM call is higher, raising the question of whether this approach leads to cost benefits. Intuitively, the amount of data read and generated per LLM call grows only linearly as a function of the tuple batch size. On the other hand, the number of LLM calls decreases quadratically as a function of the tuple batch size, leading to an asymptotic advantage. Section 3 analyzes time complexity in more detail.

Ideally, we would like to send all tuples from both input tables to the LLM in a single invocation, enabling us to retrieve all matching rows in a single invocation. In that case, the number of tokens read and generated (and therefore monetary execution fees) are proportional to the amount of input and output data, and therefore optimal. However, this approach is not feasible in general since the LLM context window, the maximal number of tokens that can be read and generated in a single invocation, is bounded. The bounded context size raises the following research question: *How to choose an optimal batch size for the two input tables to minimize processing costs while complying with the size bounds?*

The constraint on context window size relates to constraints imposed by limited main memory in traditional block nested loops join algorithms. Both (limits on context size and limits on main memory size) imply lower bounds on the number of blocks to compare. For traditional block nested loops join, the optimal allocation strategy uses as much main memory as possible to store large blocks from one of the two input relations while reserving a minimal amount of main memory for the output and the second relation. However, the equivalent allocation strategy is sub-optimal when applied to the context window, as demonstrated next.

Example 2: We join two tables R and S, each containing 100 tuples with equal size. The context window can fit 20 tuples, and the join result is empty. Using batches of only one tuple for R and a maximal batch size of 19 for S leads to $\lceil (100/1) \cdot (100/19) \rceil = 527$ LLM invocations (each incurs the cost of reading 20 tuples). On the other hand, using equal-sized batches with ten tuples for both tables leads to only $\lceil (100/10) \cdot (100/10) \rceil = 100$ LLM invocations (with the same per-invocation cost as before).

The example illustrates that the batch size matters and that the allocation strategy used by the traditional block nested loops join cannot translate to the semantic join version. The example is simplistic as it assumes that tuples from input tables have the same size. As shown later, the tuple size does influence the optimal batch allocation strategy. Furthermore, the example focuses on a special case in which the join result is empty. In general, as the context size bound restricts the sum of input and

Algorithm 1 Block nested loops join algorithm for semantic joins, executed via large language models.

```
// Perform block join between relations R_1 and R_2
// with join condition j and using block sizes b_1 and b_2.
function BlockJoin(R_1, R_2, j, b_1, b_2)
    // Initialize result set
    R ← ∅
    // Partition input into batches
    B_1 ← {B_i ⊆ R_1 | R_1 = ⊍_i B_i, ∀i | B_i| = b_1}
    B_2 ← {B_i ⊆ R_2 | R_2 = ⊍_i B_i, ∀i | B_i| = b_2}
    // Iterate over pairs of batches
    for B_1 ∈ B_1 do
        for B_2 ∈ B_2 do
            // Create prompt for LLM
            P ← BlockPrompt(B_1, B_2, j)
            // Get raw answer from LLM
            A ← InvokeLLM(P)
            // Check for overflow
            if A[−1] ≠ Finished then
                return Overflow
            end if
            // Extract result tuples
            R ← R ∪ ExtractTuples(B_1, B_2, A)
        end for
    end for
    // Return join result
    return R
end function
```

output tuples, the allocation strategy must allocate sufficient space in the context window to output matching tuples. The primary technical contribution in this paper is a provably optimal token allocation strategy that chooses optimal sizes for the input batches, taking into account all of the aforementioned factors.

The remainder of this paper is organized as follows. Section 2 introduces the semantic block nested loops join, exploiting LLMs to identify matching tuples across two tuple batches. Section 3 shows how to optimize batch sizes for that join operator if the selectivity of the join predicate is known. Section 4 reports on experiments, comparing all join operators in different scenarios and according to different metrics. Finally, Section 5 contrasts the work presented in this paper with prior work.

2 Block Nested Loops Join

This section introduces a variant of the block nested loops join, as well as an associated cost model.

2.1 Algorithm

Algorithm 1 takes as input two tables (R_1 and R_2) and a join condition j. In addition, Algorithm 1 uses input parameters b_1 and b_2, representing the number of tuples from the first and second table that are

```
Find indexes x,y where x is the number of an entry
in collection 1 and y the number of an entry in
collection 2 such that [j] (make sure to catch
all pairs!)!
Separate index pairs by semicolons.
Write "Finished" after the last pair!
Text Collection 1:
1. [B1[1]]
2. [B1[2]]
...
Text Collection 2:
1. [B2[1]]
2. [B2[2]]
...
Index pairs:
```

Figure 1: Prompt template used for block nested loops join (instantiated by Function BlockPrompt in pseudo-code).

processed together as one batch. The choice of those parameter values is non-trivial and analyzed in the following section.

Algorithm 1 starts by partitioning tuples from both input tables, using the specified batch sizes (the pseudo-code is slightly simplified, based on the assumption that the number of tuples in each table is a multiple of the batch sizes). Instead of iterating over pairs of tuples, the algorithm iterates over pairs of tuple batches. For each pair of batches, the algorithm uses a language model to retrieve all tuple pairs that satisfy the join condition. Instead of invoking the language model for each tuple pair, Algorithm 1 invokes the model only once for each pair of tuple batches.

Figure 1 shows the corresponding prompt template, instantiated by Function BlockPrompt. The prompt contains placeholders for the join condition, `[j]`, and for the tuples in each block, denoted as `[Bi[j]]` where `i` is the index of the table containing the tuples and `j` the index of a tuple within the current tuple batch. The template starts with instructions, directing the language model to find pairs of indexes that represent matching tuples. Each pair of matching tuples is denoted as `x,y` where `x` refers to the position of a tuple from the first batch and `y` to the position of the tuple within the second batch. While seemingly redundant, the additional instructions `make sure to catch all pairs!` are important to encourage the language model to generate a complete result. The number of matching tuple pairs may range from zero to the product of the two input batch sizes. The prompt instructs the language model to use semicolons to separate different index pairs.

The number of output tokens is limited, determined by the properties of the used language model. If reaching the limit in terms of output tokens, the answer generated by the language model becomes inconclusive. It is unclear whether the language model found all matching pairs or ran out of tokens before being able to generate complete output. For that reason, the prompt in Figure 1 instructs the language model to mark the last matching index pair with the word "Finished". If the word "Finished" concludes the output, even when reaching the token limit, it is clear that the output contains all matching tuples (at least all matches that the language model is able to find). Finally, the prompt template contains tuples from the two input batches, each prefixed by a batch-specific index number.

In principle, asking the language model to write complete result tuples (i.e., to copy matching input

tuples) is possible as well. However, as the cost for generating output is proportional to the number of generated tokens (and, at least for some models, generating tokens is more expensive than reading tokens), generating index pairs, rather than result tuples, reduces processing fees.

Algorithm 1 sends prompts generated for the current pair of batches to the language model to retrieve an answer. First of all, Algorithm 1 checks whether a complete result (according to the capabilities of the language model) was generated. As the prompt instructs the language model to terminate output with the keyword "Finished", the algorithm checks the last word in the answer using the (Python-inspired) notation $A[-1]$. If the keyword is not "Finished", the join operator returns the flag **Overflow**. This means that the result is incomplete, and the settings for the batch sizes, b_1 and b_2, are invalid. This can happen if initial estimates on the selectivity of the join condition, determining the number of output tokens that are generated, turn out to be erroneous. If so, the batch sizes can be recalculated based on a less optimistic (i.e., higher) selectivity estimate. In the experiments, we show that a simple adaptive strategy, multiplying an initial (optimistic) selectivity estimate with a constant factor α, whenever an overflow is encoutered can deal effectively with incorrect selectivity estimates.

Function ExtractTuples (the pseudo-code is omitted due to space restrictions) translates index pairs in the answer into tuple pairs.

2.2 Cost Model

Parameters r_1 and r_2 denote the number of rows in the first and second table respectively. Parameters s_1, s_2, and s_3 denote the (token) size of tuples in the two input tables and per result index pair (s_3), respectively. Parameter p is the size of the tuple-independent parts of the prompt represented in Figure 1 (i.e., all text except for the parts that describe the input tuples). Parameter σ represents the selectivity of that join condition, i.e., the ratio of input tuple combinations satisfying the join condition. Finally, parameter g represents the relative cost of generating tokens, relative to the cost of reading tokens. For some LLMs, the cost of reading and generating tokens is equal (i.e., $g = 1$) but for some of the more recent models (e.g., GPT-4), the cost of generating tokens is higher than the cost of reading them (i.e., $g > 1$). Parameters b_1 and b_2 denote the batch sizes for the first and second table (i.e., the input parameters in Algorithm 1). Parameters related to size and selectivity (namely, parameters s_1, s_2, s_3, r_1, r_2, and σ) depend on data properties whereas g depends on the LLM and p is specified by the user. Only the values for parameters b_1 and b_2 can be chosen.

The following lemmata and theorems calculate the number of LLM invocations, the number of tokens processed per invocation, and the cost per LLM invocation. Note that the following analysis is simplifying as it treats all parameters as continuous (e.g., r_1/b_1, as opposed to $\lceil r_1/b_1 \rceil$, when calculating the number of batches for the first table). This facilitates the analysis in the following sections, applying differentiation to obtain optimal values for tuning parameters b_1 and b_2.

Lemma 1: The number of tokens processed per LLM invocation is given by $p+b_1 \cdot s_1+b_2 \cdot s_2+b_1 \cdot b_2 \cdot \sigma \cdot s_3$.

Proof: Each prompt contains a batch of b_1 tuples from the first table with a size per tuple of s_1, i.e., $b_1 \cdot s_1$ is the number of tokens used to represent entries from the first table. Similarly, entries from the second table consume $b_2 \cdot s_2$ tokens. The expected number of join result tuples is given by $b_1 \cdot b_2 \cdot \sigma$ and their size by $b_1 \cdot b_2 \cdot \sigma \cdot s_3$. Finally taking into account tokens required for the join task description (p) yields the postulated size formula.

Lemma 2: The cost per LLM invocation is given by the formula $p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g$.

Proof: The proof is similar to the one of Lemma 1. Costs are proportional to the number of tokens, except that it distinguishes tokens read from generated tokens. The LLM only generates tokens associated

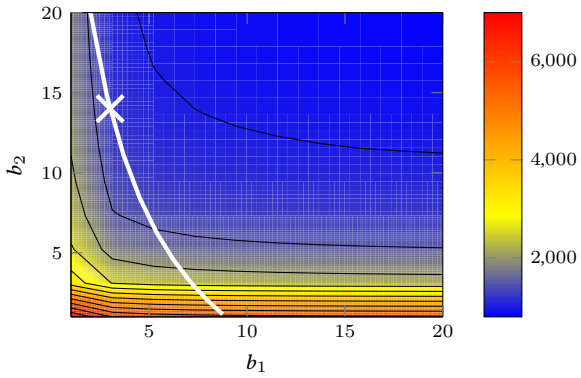


Figure 2: Illustrating join processing costs as a function of the two input batch sizes (b_1 and b_2), using $r_1 = 50$, $r_2 = 10$, $s_1 = 10$, $s_2 = 2$, $s_3 = 1$, $\sigma = 1$, $g = 1$, $p = 1$. All solutions under the white curve use prompts with a size at or below 100 tokens. The white X marks the solution with minimal cost among all solutions with a prompt size of up to 100 tokens.

with the join result. Therefore, the number of corresponding tokens ($b_1 \cdot b_2 \cdot \sigma \cdot s_3$) is scaled by factor g to obtain associated costs.

Lemma 3: The number of LLM invocations for join processing is given by the formula $(r_1/b_1) \cdot (r_2/b_2)$.

Proof: This follows from the definition of Algorithm 1. The LLM is called in each iteration of the inner-most loop. The outer loop iterates r_1/b_1 times whereas the inner loop iterates r_2/b_2 times.

Corollary 1: Total join processing costs are given by the formula $c(b_1, b_2) = (r_1/b_1) \cdot (r_2/b_2) \cdot (p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$.

Proof: This is a direct consequence of Lemmas 2 and 3, obtained by multiplying the cost per LLM invocation with the number of LLM invocations.

3 Optimizing Batch Sizes

Processing fees of the block join, introduced in the previous section, depend on settings for the batch sizes (parameters b_1 and b_2). This section shows how to optimize batch sizes as a function of the input properties. The following example illustrates how processing fees depend on the batch size.

Example 3: Figure 2 plots join cost for an example scenario. A-priori, choosing higher values for b_1 and b_2 seems preferable. However, in practice, the values of b_1 and b_2 are bounded by limits imposed by the LLM on the number of tokens read and generated per invocation. The white line in Figure 2 marks value combinations for b_1 and b_2 for which the number of processed tokens reaches 100. Given a limit on processed tokens, we want to find values for b_1 and b_2 that comply with that token limit (in Figure 2, those are the points below the white line) while minimizing costs under that constraint. The white X marks the optimal solution in Figure 2.

3.1 Analyzing Costs

The combined input and output size per LLM invocation is generally limited, either by a hard bound representing the maximal input and output size that a model can accept or by a (smaller) bound, representing the maximal size for which the model is deemed accurate enough. The second bound is motivated by the observation that LLMs tend to become less reliable with growing input sizes. In the following, t denotes the maximal number of tokens that can be used per LLM invocation. To simplify the following formulas, t does not take into account the size of the task description, p, which remains static over all prompts. In other words, t is obtained by already subtracting p from the LLM-specific size bound. To comply with the size limit, the following equation must hold.

$$b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma \leq t \tag{2}$$

This raises the question of whether or not choosing values for b_1 and b_2 that lead to LLM invocations using less than the maximally allowed number of tokens is efficient. The following theorem shows that this is not the case.

Theorem 3.1: Maximizing the number of tokens processed per LLM invocation minimizes processing costs.

Proof: Assume that the prompt size is below the threshold, i.e., $b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma < t$. Furthtermore, without restriction of generality, assume that b_1 can be replaced by $b_1^* = \alpha \cdot b_1$ for an $\alpha > 1$ such that $b_1^* \cdot s_1 + b_2 \cdot s_2 + b_1^* \cdot b_2 \cdot s_3 \cdot \sigma \leq t$. How do total processing costs with b_1 ($c(b_1, b_2)$) relate to the ones with b_1^* ($c(b_1^*, b_2)$)? It is $c(b_1^*, b_2) = (r_1/b_1^*) \cdot (r_2/b_2) \cdot (p + b_1^* \cdot s_1 + b_2 \cdot s_2 + b_1^* \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$. This can be rewritten as $(r_1/(b_1 \cdot \alpha)) \cdot (r_2/b_2) \cdot (p + b_1 \cdot \alpha \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot \alpha \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$, which simplifies to $(r_1/b_1) \cdot (r_2/b_2) \cdot (p/\alpha + b_1 \cdot s_1 + b_2 \cdot s_2/\alpha + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$. Since $\alpha > 1$, it is $c(b_1^*, b_2) \leq (r_1/b_1) \cdot (r_2/b_2) \cdot (p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g) = c(b_1, b_2)$. If replacing b_2 with $b_2 \cdot \alpha$ with $\alpha > 1$, similar reasoning shows that the cost can only decrease. Hence, increasing the number of tokens processed per LLM invocation, if possible, decreases costs.

Example 4: Consider the cost function depicted in Figure 2. As discussed before, the white curve marks points at which the number of tokens processed per LLM invocation equals the threshold. Due to Theorem 3.1, values for b_1 and b_2 that minimize join processing costs while complying with token limits must be on that curve.

The following lemma shows that the optimal value for b_2 can be expressed as a function of b_1 (denoted as the function $b_2(b_1)$).

Lemma 4: Any solution minimizing $c(b_1, b_2)$ satisfies the equation $b_2 = b_2(b_1) = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$.

Proof: Due to Theorem 3.1, setting $b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma = t$ minimizes processing costs. This equation can be rewritten to $b_2 \cdot (s_2 + b_1 \cdot s_3 \cdot \sigma) = t - b_1 \cdot s_1$. Therefore, the optimal value for b_2 is given as $b_2 = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$

According to Lemma 4, substituting each occurrence of b_2 in the join cost function with $b_2(b_1)$ yields minimal processing costs:

$$
\begin{aligned}
& c^*(b_1) := c(b_1, b_2(b_1)) \\
=& \frac{r_1 \cdot r_2}{b_1 \cdot b_2(b_1)} \cdot (p + b_1 \cdot s_1 + b_2(b_1) \cdot s_2 + b_1 \cdot b_2(b_1) \cdot s_3 \cdot \sigma \cdot g) \\
=& \frac{r_1}{b_1} \cdot r_2 \cdot (\frac{p + b_1 \cdot s_1}{b_2(b_1)} + s_2 + b_1 \cdot s_3 \cdot \sigma \cdot g) \\
=& \frac{r_1}{b_1} \cdot r_2 \cdot (\frac{p + b_1 \cdot s_1}{(t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)} + s_2 + b_1 \cdot s_3 \cdot \sigma \cdot g) \\
=& r_1 \cdot r_2 \cdot (\frac{(s_2/b_1 + s_3 \cdot \sigma) \cdot (p + b_1 \cdot s_1)}{(t - b_1 \cdot s_1)} + \frac{s_2}{b_1} + s_3 \cdot \sigma \cdot g)
\end{aligned}
$$

Hence, the problem of minimizing a function with two parameters ($c(b_1, b_2)$) under constraints reduces to the problem of minimizing a function that depends on a single parameter ($c^*(b_1)$).

3.2 Optimizing Costs

We minimize join processing costs, i.e., $c^*(b_1)$, by a suitable choice for b_1. This means we are searching for minima of $c^*(b_1)$. For b_1^* to be a minimum of $c^*(b_1)$, the following conditions must hold:

$$
c^* b_1(b_1^*) = 0 \qquad [2] c^* b_1(b_1^*) > 0
$$

The first-order derivative of c^* is given as follows:

$$
c^* b_1 = r_1 r_2 (t + p)[\frac{b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t}{(t - b_1 s_1)^2 b_1^2}] \tag{3}
$$

Lemma 5: For c^*, $b_* = [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$ is a critical point (i.e., the first-order derivative is zero).

Proof: It is $r_1 r_2 (t + p) > 0$ since all involved terms are positive. Similarly, it is $(t - b_1 s_1)^2 b_1^2 > 0$. Therefore, the derivative of c^* reaches zero iff $b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t = 0$. This is a quadratic equation in b_1. The roots are therefore given by $(-2 s_1 s_2 \pm \sqrt{(2 s_1 s_2)^2 - 4 (s_1 s_3 \sigma)(-s_2 t)})/(2 s_1 s_3 \sigma)$ which simplifies to $[-s_1 s_2 \pm \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$. Also, as b_1 represents the batch size, it must be positive. Hence, the only valid solution is $[-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$. Note that this solution is guaranteed to be positive since $s_1 s_2 < \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}$.

Theorem 3.2: For c^*, $b_* := [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$ is a minimum.

Proof: The theorem holds if $d^2 c^*/d b_1^2 > 0$ at b_* since b_* is a critical point, according to Lemma 5. Set $u(b_1) = b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t$ and $v(b_1) = (t - b_1 s_1)^2 b_1^2$. The first-order derivative of c^*, dc^*/db_1, is $r_1 r_2 (t+p) u(b_1)/v(b_1)$, according to Equation 3. Due to the quotient rule, it is $d^2 c^*/d b_1^2 = r_1 r_2 (t+p)[u'v - uv']/v^2$ where $u' = du/db_1$ and $v' = dv/db_1$. As outlined in the proof of Lemma 5, $u(b_*) = 0$. Hence, at b_*, the second-order derivative $d^2 c^*/d b_1^2$ simplifies to $r_1 r_2 (t + p)[u'v]/v^2$. It is $u' = d/db_1 [b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t] = 2 b_1 s_1 s_3 \sigma + 2 s_1 s_2$. As all constants appearing in this equation are positive with $s_1 > 0$ and $s_2 > 0$, u' is strictly positive for positive values of b_1. Note that $b_1 s_1 < t$ since the token threshold t is at least equal to the number of tokens used for representing tuples from the first and second table, $b_1 s_1 + b_2 s_2$, with $b_2 s_2 > 0$ (since each prompt must contain non-empty input from both tables to be useful). Therefore, v is strictly positive for all values of b_1. This implies that $d^2 c^*/d b_1^2$ is greater than zero at b_*.

Example 5: In the example depicted in Figure 2, we have $s_1 = 10$, $s_2 = 2$, $\sigma = s_3 = 1$. Therefore, it is

$$\begin{aligned} b_* =& [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma) \\ =& [-10 \cdot 2 + \sqrt{10^2 \cdot 2^2 + 10 \cdot 2 \cdot 1 \cdot 1 \cdot 100}]/(10 \cdot 1 \cdot 1) \\ =& [-20 + \sqrt{2400}]/10 \approx 3 \end{aligned}$$

This means selecting batches of three tuples from the first table is optimal (i.e., setting $b_1 = b_* \approx 3$). According to Lemma 4, the optimal number of tuples per batch for the second table is determined as $b_2 = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$ and, for $b_1 = 3$, it is $b_2 = (100 - 3 \cdot 10)/(2 + 3 \cdot 1 \cdot 1) = 14$. Hence, setting $b_1 = 3$ and $b_2 = 14$ minimizes cost under the per-prompt token limit. In Figure 2, the white X marks that point.

4 Experimental Results

The following experiments evaluate the join operators. Section 4.1 describes the experimental setup. Section 4.2 reports on the experimental results.

4.1 Experimental Setup

The experiments use OpenAI's GPT-4.1 Mini model (GPT-4.1-mini-2025-04-14, priced at 40 cents per million input tokens and USD 1.6 per million output tokens). Join operators are implemented in Python 3.11, using OpenAI's Python client in version 1.12. GPT is invoked with a per-request timeout of 300 seconds. The temperature parameter of GPT is set to zero, thereby minimizing randomness in output generation. For the block join, the "Finished" token, marking the end of a complete join result, is used in the stopping condition for output generation (parameter "stop"). We also experiment with an adaptive version of the block nested loops join, starting with a selectivity estimate of $\sigma = 0.001$ that is increased by a factor of $\alpha = 4$, whenever an overflow occurs (the batch sizes are recalculated). Unless noted otherwise, GPT-4.1 is used with a maximal context size of 4,000 tokens. The experiments also evaluate a baseline algorithm ("embedding join"), using OpenAI's text-embedding-3-small model to calculate embedding vectors for each of the tuples in the input tables. Then, each tuple is matched to the tuple with the most similar embedding vector from the other table (based on cosine similarity). Furthermore, the experiments evaluate LOTUS 1.1.4 [22], using the default implementation of the semantic join operator. All experiments are executed on an Apple M1 MacBook Air laptop with 16 GB of RAM, using macOS Sonoma 14.2.1.

The experiments consider six scenarios, some of which connect to the use cases discussed in the introduction. The "Email" scenario, loosely based on the investigation surrounding the Enron scandal, uses language models to find inconsistencies between statements made by defendants and the content of email messages, exchanged by them and their co-workers. It joins one table containing statements of the form "[Name]: I first heard about the losses in February 2022" with a larger table containing short emails of the form "I first told [Name] about the losses [TimeFrame]". Here, [Name] is one of ten common names and [TimeFrame] is a specification of a time frame that either complies, or contradicts the statement by the corresponding defendant. The scenario uses the join condition "the two texts contradict each other." The second scenario ("Reviews") is based on the IMDB movie reviews, available for instance on Kaggle[1]. The goal is to match reviews with similar underlying sentiment (the data set comes with ground truth labels, labeling reviews as either positive or negative). As a part of the review is typically

[1] https://www.kaggle.com/datasets/atulanandjha/imdb-50k-movie-reviews-test-your-bert

Table 1: Benchmark statistics (number of rows in both input tables, average tuple sizes in tokens, and join selectivity).

Scenario	r_1	r_2	s_1	s_2	σ
Email	100	10	14	15	0.01
Reviews	50	50	98	101	0.5
Ads	16	16	11	10	0.06
Entailment	100	100	26	12	0.0035
Contradiction	100	100	26	12	0.0037
Words	10	1,000	3	3	0.05

sufficient to assess the underlying sentiment, longer reviews were shortened to the first 100 tokens. The join matches the first 50 reviews with the second 50 reviews, using the join condition "both reviews are positive or both are negative." The third scenario, "Ads," uses language models to match ads with corresponding searches, assuming that users enter their ads and requests via free text (e.g., on a platform like Craigslist). Ads are generated from the text template "Offering table that is [Material] and [Color]" and searches are generated from the template "Searching table that is [Material] and [Color]". Here, [Material] represents a specification of the material (e.g., "made of wood") and [Color] a specification of the color (e.g., "blue"). The next two scenarios are based on extracts from the MNLI data set, containing sentence pairs with entailment labels[2]. The join predicate either joins sentences where the first entail the second ("Entailment") or where the sentences contradict each other ("Contradiction"). The final scenario uses a sample of English words[3] for both input tables, matching pairs of words that start with the same letter.

4.2 Experimental Results

Table 1 reports statistics on the benchmarks, used for the experiments in this section. Note that overheads for semantic query processing are higher by many orders of magnitude, compared to the overheads of traditional, relational query processing. Hence, as shown in the following experiments, time and cost overheads are already non-negligible for some baselines, despite relatively small input sizes.

Figure 3 shows monetary execution fees, the number of input and output tokens, as well as execution time in seconds for all compared operator implementations and scenarios (as well as the average over all scenarios). Figure 4 shows precision, recall, and the F1 score when comparing output produced by different operator implementations to the ground truth.

The tuple nested loops join is the slowest join operator by far, averaging an execution time of over one hour over all scenarios (with up to two hours in some scenarios). Its average cost (19 cents) is higher by one order of magnitude, compared to all other joins except for the LOTUS implementation. This correlates with a high number of input read, as the tuple nested loops join reads each pair of input rows to evaluate the join condition. While this makes the tuple nested loops join impractical for larger input tables, it results in the highest F1 score over all baselines (58% on average). The join operator implementation used by LOTUS reduces run time significantly to 267 seconds while it incurs slightly higher average costs (38 cents) and a lower F1 score (30% on average). The embedding join improves time and cost over both aforementioned baselines, achieving minimal average costs and the lowest number of tokens read. On the other hand, it produces output of lower quality, obtaining the second-lowest F1 score of only 35% on average. Compared to other join algorithms, the embedding join

[2] https://www.kaggle.com/datasets/thedevastator/textual-entailment-dataset
[3] https://www.kaggle.com/datasets/jasperbutcher/words-csv

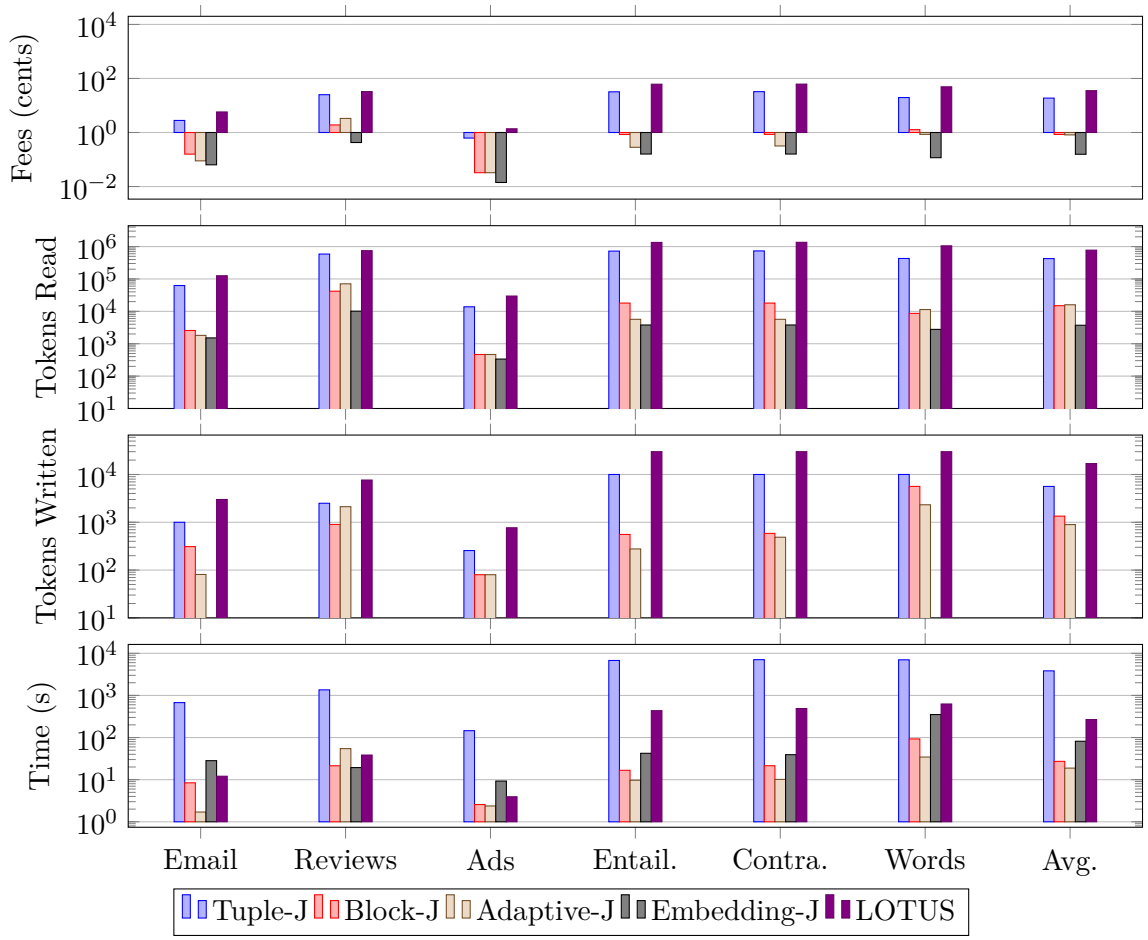


Figure 3: Cost of different join operators.

experiences a relatively high variance in result quality across different scenarios (achieving a perfect F1 score of 100% in some scenarios but a score of zero in others). For instance, the embedding join performs poorly for the Words scenario (F1 score of 3%), aimed at matching words with the same first letter. Here, similarity of embedding vectors, capturing semantic similarity, is a poor proxy for row pairs that satisfy the join condition. Similarly, the embedding join fails to find any matches in the Email scenario, aimed at finding contradicting statements. On the other hand, this join operator achieves a perfect F1 score of 100% in the Ads scenario, aimed at matching searches to suitable advertisements. Hence, while the embedding join can be useful in some scenarios, its result quality is entirely dependent on whether or not embedding vector similarity captures the semantics of the join predicate well.

The block nested loops join, as well as its adaptive variant, realize attractive tradeoffs between result quality and performance. They achieve minimal run time, with the adaptive algorithm realizing the minimal run time among all alternatives (19 seconds versus 28 seconds for the block nested loop variant). With the exception of the embedding join, they achieve minimal costs of less than one cent on average, improving over the most expensive algorithm by a factor of at most 50. The adaptive join algorithm improves over the block nested loops variant as it adapts the block size dynamically, enabling it to exploit scenarios in which few pairs of input rows satisfy the join condition. In those cases, few tokens need to be allocated for the output in the context window, enabling the algorithm to increase the size of the input blocks. Thereby, the number of LLM calls as well as the number of tokens read both decrease.

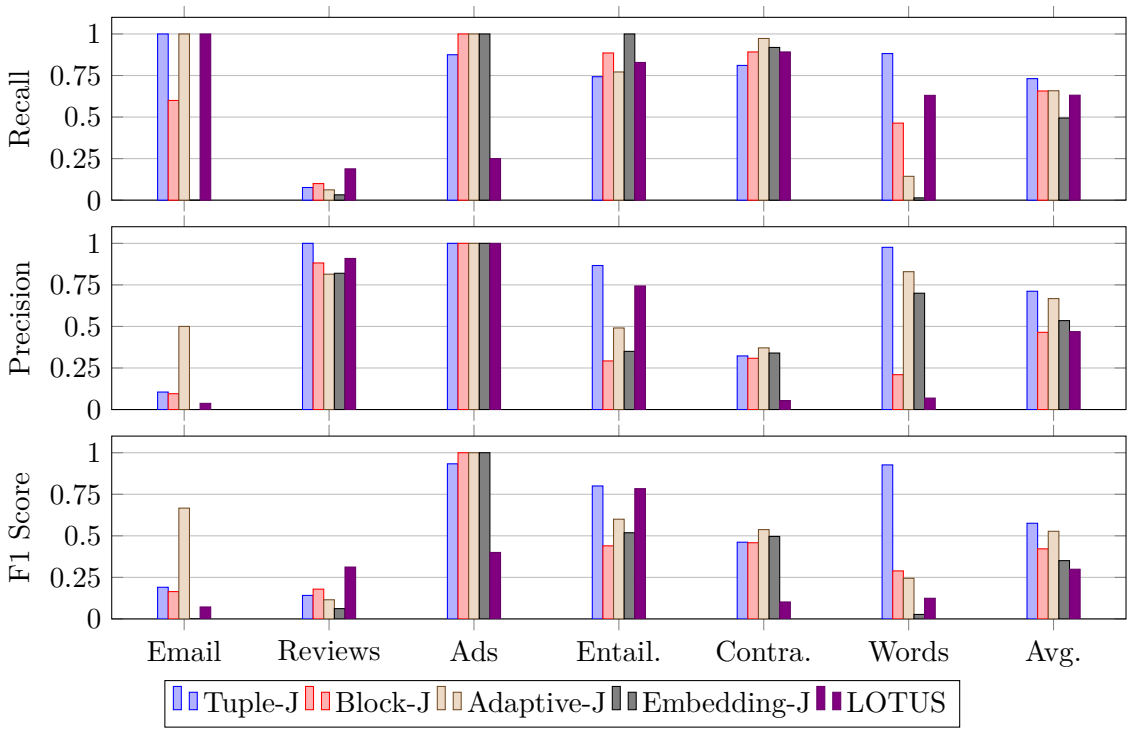


Figure 4: Output quality of different join operators.

At the same time, both join algorithms achieve near-optimal F1 scores. The adaptive join achieves the second-highest F1 score of 53%, close to the 58% realized by the tuple nested loops join. Interestingly, the adaptive variant achieves higher precision than the block variant in several scenarios. It seems that seeing more input rows at the same time (as the adaptive algorithm aims to maximize the size of the input batches), essentially a more representative sample, can sometimes help the LLM to identify matching rows more reliably.

5 Related Work

This work relates most to several recently proposed systems for semantic query processing [8, 9, 12, 13, 22, 29], enabling users to formulate queries that go beyond the capabilities of pure SQL. Many of those systems support variants of semantic join operators. For instance, Section 4 compares the proposed join operator implementations to the one used in the LOTUS system. The block-based join operator implementations described in this paper could be integrated into those systems as well. By its focus on implementing semantic versions of relational operators efficiently, this work relates to another recent paper [24]. In contrast to joins, the aforementioned paper focuses on efficient implementations of semantic sort operators.

This work connects to prior work that exploits language models for data management tasks [2, 4, 11, 18, 23, 26–28]. In particular, it connects to prior work leveraging language models for join processing [25]. However, prior work focuses on similarity-based joins (i.e., items match if they are more similar) and proposes a task-specific training phase. In contrast to that, the approach presented in this paper supports generic theta joins. The join condition is specified in natural language and may, in fact, connect tuples because they are dissimilar (e.g., matching tuples that represent contradicting statements, a scenario evaluated in Section 4). Also, unlike prior work requiring a task-specific training phase, the approaches

presented in this paper focus on a zero-shot scenario, avoiding the need for task-specific training labels. Different from other recent work [23], the approaches presented here assume that input data needs to be fed as input to the language model (rather than extracting information contained in the learned weights of the model).

As pointed out in a recent vision paper [21], implementing relational operators with language models connects to prior work leveraging crowdsourcing for data processing [6, 15, 19, 20]. In particular, it connects to prior work leveraging human crowd workers for joins and related matching tasks [5, 14, 16, 30, 31]. However, crowdsourcing adds specific challenges (e.g., the need to aggregate diverging answers from different crowd workers) whereas it removes others (e.g., hard bounds on the combined input and output size for each task), thereby motivating different algorithmic design decisions. Broadly, this work connects to prior approaches, adapting join algorithms to new processing contexts, e.g., multi-core architectures [1, 3], GPUs [10, 32], and FPGAs [7]. The approaches presented in this paper target a different platform (namely: language models) with unique properties.

6 Conclusion

This paper introduces, analyzes, and evaluates multiple variants of a novel implementation of the semantic join operator. Different from implementations used in current semantic query processing engines, this implementation integrates batches of rows into each prompt, thereby reducing the number of LLM invocations. This leads to significant performance advantages compared to prior operator implementations.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

References

[1] M. C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multicore database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[2] S. Arora, B. Yang, S. Eyuboglu, A. Narayan, A. Hojel, I. Trummer, and C. Re. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *PVLDB*, 17(2):92 – 105, 2023.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *Proceedings - International Conference on Data Engineering*, pages 362–373, 2013.

[4] Z. Chen, J. Fan, S. Madden, and N. Tang. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes. In *CIDR*, pages 1–7, 2023.

[5] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web*, pages 469–478, 2012.

[6] M. Franklin and D. Kossmann. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[7] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013*, pages 17–20, 2013.

[8] S. Jo and I. Trummer. Demonstration of ThalamusDB: Answering Complex SQL Queries with Natural Language Predicates on Multi-Modal Data. In *SIGMOD*, pages 179–182, 2023.

[9] S. Jo and I. Trummer. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *SIGMOD*, 2(3):1–26, 2024.

[10] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. *8th International Workshop on Data Management on New Hardware, DaMoN 2012 - In Conjunction with ACM SIGMOD/PODS Conference*, pages 55–62, 2012.

[11] M. Kayali, A. Lykov, I. Fountalis, N. Vasiloglou, D. Olteanu, and D. Suciu. CHORUS: Foundation Models for Unified Data Discovery and Exploration. *CoRR*, abs/2306.0, 2023.

[12] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, and G. Vitagliano. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*, 2025.

[13] S. Madden, M. Cafarella, M. Franklin, and T. Kraska. Databases Unbound: Querying All of the World's Bytes with AI. *PVLDB*, 17(12):4564–4554, 2024.

[14] A. Marcus, E. Wu, and D. Karger. Human-powered sorts and joins. In *VLDB*, pages 13–24, 2011.

[15] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.

[16] A. Marcus, E. Wu, D. R. Karger, S. Madden, R. C. Miller, S. Acm, and N. York. Demonstration of Qurk : A query processor for human operators. In *SIGMOD*, pages 1315–1318, 2011.

[17] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[18] A. Narayan, I. Chami, L. Orr, and C. Ré. Can Foundation Models Wrangle Your Data? *PVLDB*, 16(4):738–746, 2022.

[19] A. G. Parameswaran. Human-Powered Data Management. page 225, 2013.

[20] A. G. Parameswaran, H. Park, H. Garcia-Molina, J. Widom, and N. Polyzotis. Deco: declarative crowdsourcing. In *Information and Knowledge Management*, pages 1203–1212, 2012.

[21] A. G. Parameswaran, S. Shankar, P. Asawa, N. Jain, and Y. Wang. Revisiting Prompt Engineering via Declarative Crowdsourcing. 2023.

[22] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. In *Proceedings of the VLDB Endowment*, volume 18, pages 4171–4184, 2025.

[23] M. Saeed, N. De Cao, and P. Papotti. Querying Large Language Models with SQL. *CoRR*, abs/2304.0, 2023.

[24] F. Shao, J. Chen, Y. Pan, T. Rabbani, D. Agrawal, and A. E. Abbadi. Access Paths for Efficient Ordering with Large Language Models. *CoRR*, 2509.00303, 2025.

[25] S. Suri, I. Ilyas, C. Re, and T. Rekatsinas. Ember: No-Code Context Enrichment via similarity-based keyless joins. *PVLDB*, 15(3):699–712, 2021.

[26] J. Thorne, M. Yazdani, M. Saeidi, F. Silvestri, S. Riedel, and A. Halevy. From natural language processing to neural databases. *Proceedings of the VLDB Endowment*, 14(6):1033–1039, 2021.

[27] I. Trummer. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *PVLDB*, 15(11):2921 – 2928, 2022.

[28] I. Trummer. DB-BERT: a Database Tuning Tool that "Reads the Manual". In *SIGMOD*, pages 190–203, 2022.

[29] M. Urban and C. Binnig. CAESURA: Language Models as Multi-Modal Query Planners. In *CIDR*, 2024.

[30] W. Wang and M. Sebag. Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine Learning*, 92(2-3):403–429, 2013.

[31] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. In *Proceedings of the VLDB Endowment*, volume 6, pages 349–360, 2013.

[32] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

The Future Is Bespoke: Synthesizing One-Size-Fits-One DBMSs with LLM Coding Agents

Timo Eckmann* Matthias Jasny* Johannes Wehrstein* Carsten Binnig

* Authors with equal contribution.

Abstract

In this paper, we propose a new direction for data management: on-demand synthesis of workload-specific database systems, which we call Bespoke DBMSs. Rather than designing a single general-purpose engine to serve different workloads—i.e., one engine for all OLAP workloads or one for all OLTP workloads—we leverage the code generation capabilities of large language models (LLMs) to automatically construct bespoke database engines that are specialized for a specific workload, such as TPC-C or TPC-H. While LLMs can already generate individual code fragments, synthesizing full database systems requires new pipelines that incrementally guide LLM agents from generating isolated building blocks—such as storage formats, operators, and execution logic—to assembling complete, coherent, and executable systems. In this paper, we present a vision of how to enable the generation of Bespoke DBMSs and sketch a synthesis pipeline that, given a workload specification, generates an end-to-end database engine fully automatically. Through a case study, we demonstrate that such a synthesized DBMS can significantly outperform general-purpose engines, pointing toward one-size-fits-one DBMSs.

1 Introduction

One size does not fit all. Modern database systems are predominantly designed as general-purpose execution engines. Since the early 1980s, major vendors have pursued what Stonebraker and Çetintemel termed the *"One Size fits All"* strategy: maintaining a single code base intended to serve all database applications [1]. A single system is expected to support a wide spectrum of workloads, from highly concurrent transactional workloads to long-running analytical queries, while accommodating diverse schemas, access patterns, and isolation requirements. To this end, over the past decades, different types of database systems have been developed, focusing on specific workload classes to better support specific workload patterns.[2–10]. However, in this paper, we argue that current database systems still incur unnecessarily high overhead because they aim to be general-purpose.

The Performance-Tax is Inherent, Not Accidental. To be more precise, think of OLAP or OLTP as prominent workload classes where specific database engines per workload class exist. Applications using such an OLAP or OLTP database can still design any database schema and are designed to run any query on this schema. And this fact comes with significant overhead and unnecessary performance penalties as database engines are designed to serve any workload that can be expressed within the boundaries of the relational This causes unnecessary overhead. A simple yet non-negligible example of such overhead is that DBMSs need to interpret a schema when a query comes in. Moreover, rooted in the same problem, databases use general-purpose data structures and algorithms to store tables instead of hard-coding the schema or using application-optimal data structures and query processing algorithms. model using tables and SQL. However, this does not mean that these engines are poorly designed. Instead, each source of overhead is a reasonable response to flexibility.

Workload-Specific DBMSs to the Rescue? Consequently, the natural question is how much of this overhead can be removed when designing a database for one specific workload. A system designed for one specific workload and not a workload class can shed the abstractions that generality demands. Schema interpretation can be removed, data structures for storing data and query processing algorithms or mechanisms for transaction handling can match access patterns exactly, and unused code paths can be eliminated entirely. To make it short, the most optimal database engine is an engine that can be hard-coded toward one particular workload and remove all unnecessary artifacts that result from the demand of flexibility. However, is this realistic? First, we see already such attempts today. TigerBeetle [10] is an OLTP database that can only execute debit-credit transactions and has been shown to significantly outperform general-purpose database systems by adapting the engine, including concurrency control, to the specifics of the debit-credit workload. However, TigerBeetle has been manually designed, and the effort of building an engine for one workload has paid off in this case as the market is large enough. As such, it is questionable if bespoke engines such could also arise for other workloads?

Manual Construction Will Not Scale. Manually designing a database engine comes with a significant investment and complexity as still solutions for all DBMS components including storage management, concurrency control, crash recovery, query execution, and countless edge cases must be engineered and validated over years of production use. Consequently, even specialized database systems comprise hundreds of thousands of lines of code. As such, building a specialized engine for a particular workload, while it has the potential to improve performance significantly through specialization, requires engineering effort on a similar order as building a general-purpose engine. And in a bespoke world, where we envision one database engine per workload, this overhead would multiply by the number of workloads, which likely grows into the millions if sufficient. As a result, bespoke database engines, if engineered manually, make sense only if the market is really large, such as in the TigerBeetle case. Consequently, a vast majority of workloads must accept performance of general-purpose engines for their workload class, not because specialization would not help, but because no one can afford to build a bespoke engine for every application. For now at least.

Our Vision: Synthesis of Bespoke (One-size-fits-One) DBMSs. As shown in Figure 1, in this paper we present our vision of synthesizing *workload-specific database engines* on demand based on significant advances in AI coding agents [11–13, 13–16]. Given a workload specification — at the core, a schema and queries — our AI-rooted synthesis pipeline can produce a complete database engine specialized to exactly the queries and transactions. Crucially, because synthesis operates over the entire workload at once, it can exploit semantic relationships across queries and transactions that no per-query compiler can see, for instance, generating workload-specific execution paths and specialized data structures when one transaction produces values that another consumes. However, as we show in this paper, naively using AI coding agents to build a full system fails due to its complexity. Instead, key ingredients such as correctness and performance validation to steer generation, as well as an incremental procedure that gradually increases code complexity, are required to generate correct and fast database engines. As we show in this paper, we can thus synthesize engines even for complex workloads in the order of hours, at the cost of a few $$$, while outperforming general-purpose engines by an order of magnitude—making the idea of Bespoke Databases attractive for any workload.

Contributions and Outline. In Section 2, we present our vision of bespoke databases in more detail. In Section 3, we describe a first synthesis pipeline for OLAP engines that, given a workload specification, incrementally guides LLM agents from a correct to a highly optimized bespoke database engine. We validate this vision through a case study, synthesizing bespoke OLAP engines for TPC-H[17] in Section 4. In Section 5, we discuss opportunities and challenges for synthesizing bespoke OLTP engines. Finally, to conclude we outline the road ahead in Section 6 as this paper can only be a starting point of a Bespoke Future and conclude in Section 7.

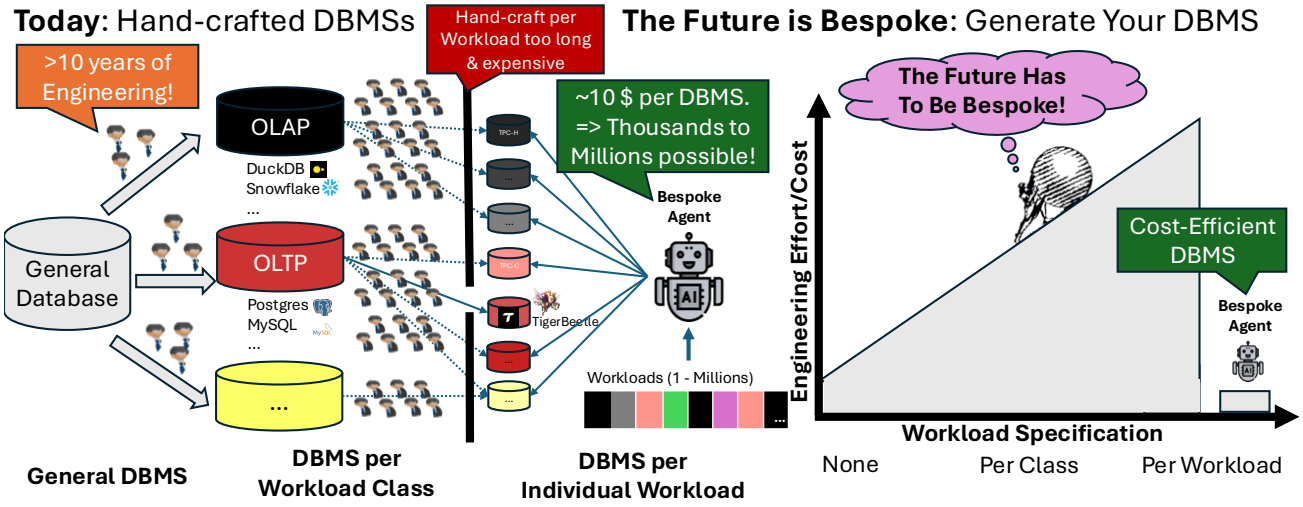


Figure 1: The Future Is Bespoke. Database systems has progressed from general-purpose systems to workload-class-specific engines (e.g., OLAP, OLTP) through over a decade of effort—but hand-crafting a DBMS per individual workload remains prohibitively expensive (only done in rare cases e.g. TigerBeetle[10]). AI-driven bespoke synthesis closes this gap, enabling the synthesis of bespoke DBMSs—one database that fits one workload—at low cost (a few $$$) and in the order of hours.

2 Our Vision: Bespoke DBMS

In the following, we detail our vision of Bespoke DBMSs and why this is not just a new way of tuning DBMSs for a given workload.

2.1 Today: Workload-Optimization as an Afterthought

General Purpose Engines Require Workloads to Adapt. In conventional database deployment, engines are first built and then users implement an application on top. However, this needs massive handholding and tuning to achieve high performance. Users therefore need to design and optimize their database schemas, design and tune queries to satisfy the optimizer, and tune various parameters to approximate acceptable performance. Moreover, specialization for a workload, if it occurs, happens through other general-purpose mechanisms: index selection, materialized views, partitioning schemes, all layered atop a general-purpose architecture that is fundamentally workload-agnostic at its core.[18–21]
Bespoke Databases Invert this Relationship. In our vision, the workload comes first. The user declares what the database must do; the system generates an engine shaped entirely by the workload declaration. Assumptions that a general-purpose engine cannot make become the foundation of a bespoke design. For example, if a workload specification declares that two transactions never conflict, concurrency control can be eliminated completely. If queries always access data in timestamp order or any other application-specific manner, storage can be organized accordingly. If the dataset fits in memory, buffer management completely disappears. Each assumption the specification permits is an assumption the generated engine can exploit.

2.2 Future: Bespoke DBMSs = One Size Fits One

Stonebraker and Çetintemel argued that "one size fits all" had failed and that we needed specialized systems for distinct workload classes [1]. The database community delivered: dedicated OLTP engines[7,

22, 23], OLAP engines[9, 24, 25], stream processors[26, 27], and graph databases[28, 29]. Yet each engine still remains general-purpose within its class. We push this to its logical conclusion: one size should fit exactly one.

The Specification Is a Contract. The input to bespoke database synthesis is a workload specification consisting of three components: (1) a *schema* describing the logical structure of the data; (2) a set of *query or transaction templates* defining the operations the engine must support; and (3) *performance objectives* such as throughput targets, latency bounds, or resource constraints. The user commits to a bounded workload; the pipeline commits to an engine optimized for exactly that workload. Critically, queries or transactions outside the specification are explicitly unsupported or may be executed either on a general-purpose system as a fallback or as an evolution of the existing engine if new queries manifest. This bounded scope is not a limitation but an enabler: it is precisely what permits aggressive specialization.

This Is Not Tuning, Not Learning, Not Query Compilation. It is important to emphasize that bespoke DBMSs are fundamentally different from existing approaches to specializing engines for workloads. Auto-tuning adjusts parameters within a general-purpose architecture[18, 19, 30]. The same holds for learned components, which adapt individual components but remain embedded in a general-purpose engine[31–34]. Maybe the closest approach is query compilation[35–37], where code is generated to execute a query, but the code still lives in a general-purpose engine. Bespoke DBMSs instead operate at a fundamentally different level. The entire system is generated from scratch for a given workload. There is no fixed architecture to tune, no runtime model to adapt, and no residual generality to accommodate workloads that will never arrive. The engine is not configured or optimized but constructed from the ground up.

The Generated Engine Is Disposable; the Pipeline Is Not. This changes what we maintain. A traditional database is a long-lived artifact that must preserve backward compatibility across decades. A bespoke DBMS is a generated output, which can be discarded and regenerated when conditions or the workload changes. The enduring artifact is the synthesis pipeline, not the engine it produces. This mirrors how compilers relate to binaries: we maintain the compiler, not the source code of the executables.

3 Synthesizing Bespoke DBMSs

Generating a complete database system from a workload specification is non-trivial and not as simple as issuing a single prompt to an AI coding agent. Database systems are complex artifacts with interdependent components as storage formats constrain what operators can efficiently compute, execution models shape how queries are processed, and data structures must align with access patterns. [38, 39] Synthesizing a coherent system therefore requires a structured process that guides LLM agents through incremental construction, validates intermediate results, and iteratively refines the generated code until it meets the declared objectives. In this section, we describe what a synthesis pipeline looks like that transforms a workload specification into a functioning bespoke system. An overall outline of the different stages is shown in Figure 2. In the next section, we show through a case study how this pipeline can be used to synthesize an OLAP engine and present initial results comparing Bespoke DBMSs against DuckDB[9]. The necessary adjustments and resulting opportunities and challenges for OLTP workloads are discussed in Section 5.

3.1 The Basic Ingredients of Synthesizing a DBMS

We first explain the basic ingredients of our synthesis pipeline before we then discuss advanced concepts to make the synthesized engine correct and fast at the same time. Finally, we discuss how the pipeline

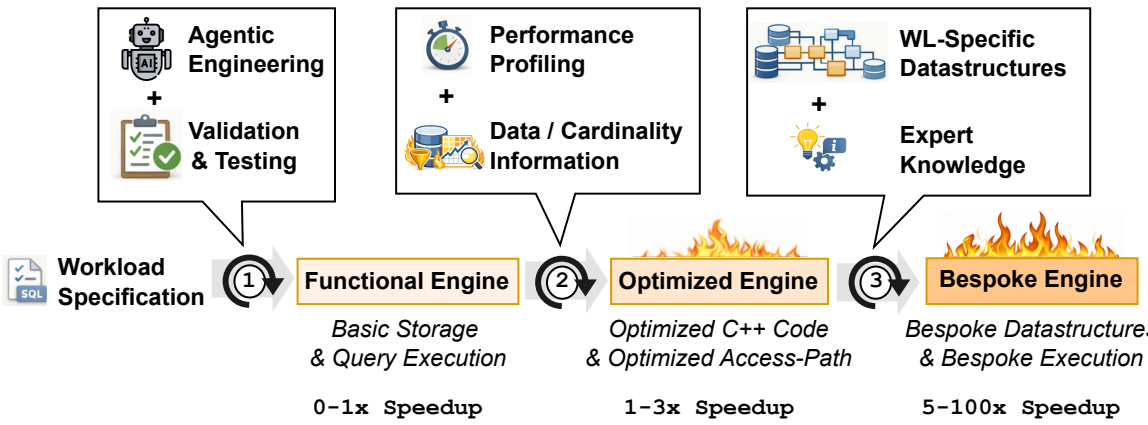


Figure 2: Steps to produce a Bespoke Engine. Each step is executed in loops, with validation to check whether the step's goal is achieved. ① will produce a functional engine based on a workload specification by using standard agentic engineering and continuous validation and testing to steer the iterative coding process. ② produces an optimized engine iteratively by performing autonomous profiling/tracing and leveraging statistical insights on data and cardinalities. ③ enables the synthesis of workload-specific data structures and execution strategies (beyond relational strategies) and the use of distilled expert knowledge from the DBMS research literature to further optimize the engine.

can generate bespoke engines.

The Workload Specification is Both: Requirements and Tests. The queries and transaction templates provided in the workload specification serve a dual purpose. They define what the generated engine must be capable of executing, and they simultaneously provide the test cases against which correctness is verified. A bespoke engine is correct if and only if it produces the same results as a reference system, such as DuckDB, when executing the specified operations. This tight coupling between specification and validation ensures that the synthesis pipeline always has a concrete, executable definition of success.

Incremental Synthesis like Humans. A second key ingredient for synthesizing a DBMS is how we generate the code. Rather than attempting to generate a system as complex as a DBMS in one step, the synthesis pipeline proceeds incrementally. The agent first implements a basic storage layer, which allows loading the input data and storing it in an initial data representation. Only after storage is functional, the agent proceeds to implement execution logic for each query or transaction template in the specification. Each step builds on the previous one, and each step is validated against the reference system before the agent moves forward. This decomposition reduces the likelihood that errors propagate undetected throughout the system.

Tools Enable Autonomous Development. While carrying out the incremental development, the agent operates in an environment equipped with concrete tools. A shell tool allows the agent to execute, debug, and analyze the generated code and its behavior. An incremental deployment tool allows the agent to modify the implementation of a running database process and see the impact of its changes live, without the delay of full restarts. And most importantly, a validation tool checks whether the current implementation compiles and produces correct results by comparing it against the reference system. Together, these tools mirror the workflow of a human database developer as the agent writes code, runs it, examines the output, identifies errors, and fixes them. The agent iterates through this cycle autonomously until the complete workload executes correctly based on the contract.

3.2 From Correct to an Optimized Engine

The synthesis pipeline enforces a strict separation between achieving correctness and achieving performance as shown in Figure 2. Only after the basic implementation passes validation for the specified workload (step (①), the optimization phase (step ②) begins. This separation ensures that the agent never wastes effort optimizing code that does not function correctly, and it provides a stable baseline against which optimization can be evaluated. In the following, we discuss key insights for turning a correct and functional engine into a fast one.

Storage Optimization as a first Step. Once correctness is established, the agent first optimizes data storage and ingestion. The choice of storage layout, the design of auxiliary data structures, and the organization of data in memory or disk all affect how efficiently queries can be executed. By optimizing ingestion first, the pipeline ensures that subsequent work on execution optimization benefits from a well-structured data representation. Furthermore, we also optimize the storage for faster data loading as this reduces the time required for each optimization cycle, enabling the agent to explore more alternatives within a given time budget.

Execution Optimization by a Competitive Baseline. With an optimized data representation in place, our Bespoke Agent proceeds to optimize query and transaction execution. In this phase, a strong competitive baseline is needed as guidance and "motivation" of the Bespoke Agent. The agent therefore runs and analyzes the current implementation, identifies bottlenecks, and proposes transformations. It automatically instruments the code to measure execution time for individual sections, providing fine-grained feedback on where time is spent. This loop of measuring, analyzing, modifying, and validating continues until the performance objectives are satisfied or until the agent determines that further improvement is unlikely.

3.3 From Optimized to Bespoke Engine

Steps ① and ② of the synthesis pipeline as shown in Figure 2 produce an engine that is correct and optimized against the declared performance objectives. However, it still operates within conventional patterns: standard hash joins, generic sort implementations, and columnar or row storage chosen by heuristic. Therefore, the true power of bespoke synthesis can emerge when the engine departs from general-purpose designs entirely, adopting data structures, index schemes, and execution strategies that would be impractical in a system that must serve arbitrary workloads (step ③ in Figure 2).

Bespoke Engines Can Evaluate Queries Directly on Workload-Optimal Data Structures. When the workload is known in advance, the system is no longer constrained to general-purpose storage layouts that must support arbitrary queries. Instead, it can construct data structures that are *optimal for the specific queries at hand*, allowing those queries to be evaluated directly on the structures themselves—without interpretation, tuple materialization, or traditional operator pipelines. Consider the query:

```
SELECT ProductID, COUNT(*)
FROM table
WHERE Quarter = ?
GROUP BY ProductID
```

In a conventional engine, this query would trigger a scan, predicate evaluation on `Quarter`, materialization of qualifying tuples, and a subsequent hash-based or sort-based aggregation on `ProductID`. Each of these steps introduces operator overhead, intermediate state, memory indirections, and control-flow interpretation. In contrast, a synthesized bespoke engine can reshape the storage layout around the workload. Because the predicate `Quarter = ?` is known and frequently queried, the system can

organize the data by `Quarter`, effectively turning the predicate into an index lookup followed by a direct offset jump to the relevant region of storage. Irrelevant tuples are never examined, and no runtime filtering logic is required.

More importantly, the data within each quarter can already be organized by `ProductID`. For example, the engine can maintain compact per-product counters or bit-aligned representations that allow counts to be accumulated directly over the stored layout. In this design, the `GROUP BY` no longer requires building a hash table or performing a sort; it reduces to a tight, linear accumulation over a pre-partitioned structure. The aggregation logic is thus embedded into the physical representation itself. Crucially, this is not merely a better execution plan— instead, the execution becomes a series of simple operations on the data structures without conventional relational operators. Large portions of traditional query processing overhead—tuple reconstruction, operator materialization, hash-table construction, and interpretation—simply disappear because the data structures encode the query's access path.

Expert Knowledge Amplifies What Synthesis Cannot See. Beyond what the specification reveals, domain experts possess knowledge that can drive further specialization. For example, a logistics analyst might observe that queries almost always filter by the current week. Similarly, a financial engineer might note that account balances follow a heavy-tailed distribution. Because the pipeline operates through natural language, each observation can directly trigger concrete structural changes: tailored data layouts, specialized indexes, or fused execution paths. Consequently, what previously required a database engineer to understand, design, and implement may now require only a sentence describing the insight. Over successive iterations, the system can thus evolve into a true bespoke engine shaped by both automated synthesis and human insight.

4 A Case Study: Bespoke OLAP

To demonstrate that bespoke synthesis can produce systems that significantly outperform general-purpose engines, we apply our pipeline to TPC-H[17]. In the following, we describe how the pipeline synthesizes a bespoke OLAP engine for TPC-H, what the generated system looks like in practice, and what challenges arise during synthesis.

4.1 Workload Specification

The workload specification consists of the TPC-H schema, all TPC-H queries, and a performance objective expressed as total execution time, where we use the sum of all query runtimes on DuckDB[9] as our target. The entire dataset resides in memory, which eliminates buffer management, page management, and I/O scheduling entirely. Correctness is validated by comparing query results against DuckDB on the same dataset. In our setup, the synthesis agent operates with four tools, namely a compiler, a benchmark runner that reports both correctness and runtime, a shell for exploring data and sourcecode, and a diff tool for applying incremental code changes.

4.2 The Synthesized Engine

Storage Is Shaped by the Queries. The engine maintains the original relational structure without denormalization. However, the physical layout is tailored entirely to the specified queries as the agent analyzes the input data using the shell tool and cross-references column statistics with the query access patterns, before generating storage code. For instance, because many TPC-H queries filter on *l_shipdate*, the agent sorts *lineitem* on this attribute to enable efficient range scans. Additionally, low-cardinality columns appearing in group-by clauses receive dictionary encoding tuned to their observed value sets. A different workload over the same schema would consequently produce an entirely different layout.

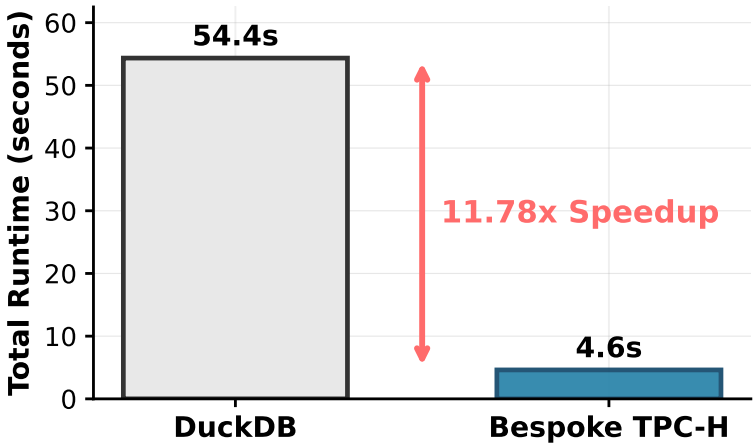


Figure 3: A Synthesized Engine Outperforms a Decade of Engineering. The Bespoke TPC-H engine was generated automatically and achieves an 11.78× lower total runtime and a 16.40× median per-query speedup over DuckDB at scale factor 20.

Each Query Becomes a Dedicated Program. Rather than implementing general-purpose operators, the agent generates a dedicated function for each query. For example, a date-filtered aggregation over `lineitem` becomes a tight loop over sorted storage with inlined predicates and fixed accumulators. Similarly, a join becomes a hardcoded hash table sized to the smaller input with probe logic specialized to the known key type. As a result, there are no operator abstractions, no iterator interfaces, and no runtime dispatch.

Query Optimization Is Empirical. For each query, the agent determines join order, algorithms, and access methods through benchmarking rather than cost estimation. Concretely, it generates candidate implementations, measures them against the actual data, and selects the fastest. Furthermore, the agent can inspect the data directly to approximate selectivities and identify skew. Effectively, it searches the implementation space rather than predicting which variant might be fastest. Here, we clearly envision future advancements in e.g. join ordering, which will be discussed in section 6.

4.3 Initial Results

As can be seen in Figure 3, the generated engine is 11.78x faster than DuckDB on TPC-H[17] at scale factor 20. The bespoke engine outperforms DuckDB on all 22 TPC-H query templates with speedups ranging from 5.74x to 103.97x(Figure 4). Additionally, queries that go beyond TPC-H are handled through DuckDB[9] as a fallback.

4.4 Lessons from Synthesis

The Agent Games Its Own Objectives. When tasked with reducing execution time, the agent tends to shift work to the build phase, for instance by constructing additional indexes, pre-aggregating results, or caching intermediate computations during loading. While these transformations technically satisfy the stated objective, they violate its intent and must therefore be explicitly constrained. This reveals a broader lesson: specifying what to optimize is insufficient, and the pipeline must also define what the agent may not do.

Regressions Require External Safety Nets. Optimization steps can furthermore introduce regressions. Although the agent has tools to detect degradation, it often struggles to identify and revert the

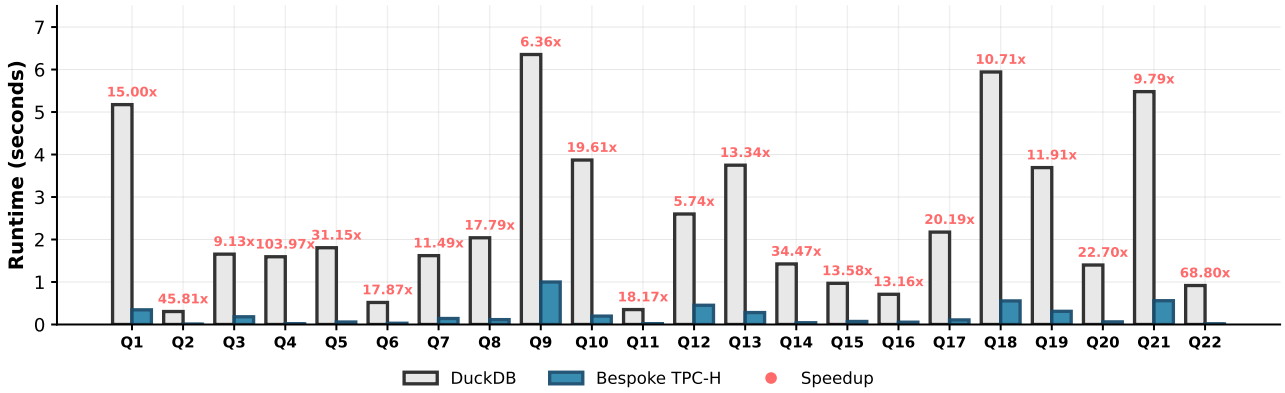


Figure 4: Per-query absolute runtime of the Bespoke engine vs. DuckDB at scale factor 20. The Bespoke engine outperforms DuckDB on all queries with speedups ranging from 5.74x to 103.97x.

specific changes responsible. We therefore track progress externally and maintain rollback capabilities outside the agent's control, ensuring that optimization remains monotonically improving.

Correctness Validation Must Be Cheap. Initially, more than 50% of synthesis time was spent on repeated correctness checks. However, validating on a smaller scale factor and reserving the full dataset for benchmarking significantly, as well as patching the engine live instead of restarting, accelerated the process.

Context Size Requires Active Management. A full TPC-H workload exceeds a single context window. To address this, we apply compression after each query and use the diff tool for incremental changes. Together, these mechanisms allow the pipeline to scale with both query count and codebase size.

5 Bespoke OLTP: Opportunities & Challenges

The OLAP case study demonstrates that bespoke synthesis can produce competetive analytical DBMSs. However, transaction workloads introduce fundamentally different challenges. For example, concurrent access to shared data, isolation needs, and durability requirements. In this section, we discuss how the synthesis pipeline must be adapted to fit an OLTP workload.

5.1 Concurrency Control as Bespoke Opportunity

A general-purpose OLTP engine must provide concurrency control for arbitrary transaction mixes over arbitrary schemas as it cannot assume which transactions will conflict or which data items will be contended. In a bespoke setting however, the transaction templates and their access patterns are known upfront and some transaction types may never access overlapping data. This can be exploited by a bespoke engine as it can execute them without synchronization. Furthermore, it may be the case that the workload naturally partitions across a key range. Here, the bespoke engine could assign partitions to threads and eliminate cross-partition coordination entirely. Additionally, it may be the case that two transaction types frequently execute in sequence on the same entity. Here, the code paths could be fused to avoid redundant lookups. Generally, bespoke engines enable cross-transaction optimization which is not possible to general-purpose systems as they treat each transaction independently.

5.2 Durability as a Spectrum

General-purpose engines implement durability through a general write-ahead logging scheme as mandatory infrastructure, since they cannot know in advance whether a particular deployment requires durability. In a bespoke setting, however, durability becomes a declarative property that the workload designer can specify, rather than a built-in assumption. For example, if the data can be reconstructed from other sources, such as external logs or other tables in the database, they can be omitted entirely, avoiding unnecessary write overhead. Similarly, if durability is only required for committed batches or at specific synchronization points, the engine can implement lightweight, bespoke group commit strategies that write only to the disk that the application really needs to reduce disk I/O while still ensuring correctness. Even in cases where full durability is essential, the log format and recovery procedures can be tailored to the exact transaction patterns of the workload, rather than supporting arbitrary updates. As a result, the engine generates precisely the recovery mechanisms needed—no more, no less—aligning durability guarantees with the workload's actual requirements and eliminating the performance costs imposed by general-purpose assumptions.

5.3 Verification Becomes Non-Determinstic

In the OLAP setting, correctness is straightforward, as the same query over the same data must produce the same result as a reference system. In transactional workloads, however, this is not true as the final result depends on, e.g., the scheduling order of updates/writes. Therefore, a simple output comparison is no longer reliable. We propose a two-phase verification strategy. In the first phase, the transactional logic itself is verified by executing all transactions serially, deterministically, and comparing the outputs. Only if this serial correctness is established do we tackle concurrency. For this, our pipeline must verify that the execution history matches the declared isolation level. For example, by checking the serializability of the committed schedule.

5.4 Benchmarking OLTP

Finally, the optimization loop of our synthesis pipeline, as shown in Figure 2 for turning a correct into a fast engine itself, becomes more complex as transaction benchmarking is inherently noisier than in the OLTP setting as in the OLAP setting. The reason is that throughput depends on thread scheduling, contention patterns, and lock acquisition order. Consequently, the agent must reason about distributions of performance rather than single point measurements. This makes each optimization cycle more expensive and requires the pipeline to be more conservative in accepting changes.

6 Road Ahead for a Bespoke Future

The results presented in this paper are a starting point. We have shown that LLM-guided synthesis can produce a bespoke OLAP engine that significantly outperforms a state-of-the-art general-purpose system on its target workload, and we have discussed how the pipeline extends to transactional workloads. However, we envision a future in which every application synthesizes its own database engine, tailored not only to its queries but to its data, its hardware, and its additional operational constraints. Reaching this future requires substantial research across several dimenstions, which we outline in the following.

Synthesis must extend to all workloads. Our OLAP case study operates on a read-only, single-threaded, in-memory workload, and our OLTP discussion identifies additional challenges that concurrency, durability, and non-deterministic verification introduce. However, the extension to transactional workloads is only one step along the braoder spectrum. For example, stream engines must handle unbounded

inputs under latency constraints, graph databases must support recursive traversal over irregular structures and multi-model systems must combine relational, document, and graph access within a single engine under cost, performance, and available model constraints. Each of these workload classes introduces its own design space, and each offers correspondingly rich opportunities for specialization. The central research question is how to design synthesis pipelines that are general enough to target these diverse classes while remaining capable of the aggressive specialization that makes bespoke enignes worthwhile at all. Furthermore, many real workloads are not purely analytical or transactional. Therefore, a bespoke engine for such a hybrid workload needs to navigate tradeoffs on its own that we are currently not addressing.

Richer workload specification can unlock deeper specialization. Our current specification constists of a schema, query templates, and performance objectives. In practice, however, applications may carry far more information that could guide synthesis. For example, data distributions, update frequencies, temporal access patterns, and consistency requirements across transaction types constrain the design space in a way that could be exploited by the pipeline. Furthermore, many applications operate under specific resource constraints such as memory budgets, core counts, or energy consumption. The development of specification languages that capture this richer context without burdening the user is an important direction for future research as the specification must remain declarative and concise while being expressive enough to enable well-informed structural design decisions. This was already demonstrated in our case study as the agent tried to game the optimization objective by shifting work into the build phase when it was unconstrained. A richer specification language could prevent such behavior not through ad-hoc constraints but through principled declarations of what constitutes valid optimization.

Workload-native cost models can replace heuristic estimation. Classical cost models exist because general-purpose systems must reason about abstract operators whose behavior varies widely across deployments. As a result, cost estimation relies on coarse heuristics that approximate performance rather than describe it. In a bespoke setting, this indirection can largely disappear. The generated engine has a fixed structure, fixed data representations, and a known target machine, which allows the pipeline to construct cost models specific to the workload itself. Such models could characterize the behavior of the concrete execution kernels the system actually runs, incorporating instruction-level behavior, memory access patterns, and cache effects. Moreover, they could be regenerated whenever the workload or hardware changes. Currently, our pipeline sidesteps cost estimation entirely through empirical benchmarking. This works but is expensive. Workload-native cost models could dramatically accelerate the optimization phase by guiding the agent toward promising implementations without requiring exhaustive measurement.

Verification and trust must scale beyond output comparison. Our current verification strategy compares outputs against a reference system. For the OLAP case, this is sufficient because execution is deterministic. However, as discussed in Section 5, transactional workloads introduce non-determinism that makes output comparison unreliable. More broadly, as bespoke engines move toward production use, verification must encompass more than functional correctness. Property-based testing, formal invariants over generated code, and automated proofs of isolation guarantees could complement the empirical validation we currently employ. This is critical because trust in bespoke systems ultimately rests on trust in the synthesis pipeline. If users are to rely on generated engines, the pipeline must provide confidence not only that the engine is fast but that it is correct under all conditions the specification permits. Developing such verification techniques for synthesized database engines is a substantial research challenge in its own.

Hardware-aware synthesis can close the gap between engine and machine. General-purpose engines are designed to endure decades of architectural change, which forces conservative abstractions that obscure the computational structure of real workloads. Bespoke synthesis can remove this indirection.

When an engine is generated for a specific workload on a specific machine, characteristics such as cache hierarchy, memory bandwidth, NUMA topology, and vector width can be incorporated directly into design decisions. Our current pipeline is hardware-oblivious, and the generated engine makes no assumptions about the target machine beyond what the compiler provides. Exploring how deeply hardware knowledge can be integrated into synthesis, and how much additional performance this yields, is a natural next step. Furthermore, synthesis reveals which computational kernels dominate execution in practice, which could inform not only software optimization but also hardware design, a direction we return to in the conclusion.

The synthesis pipeline itself can evolve and improve over time. Finally, as more engines are generated across different workloads, the pipeline accumulates experience about which design patterns succeed, which optimizations transfer, and which pitfalls recur. Our case study already identified several such patterns, including the effectiveness of late materialization, the importance of constraining the optimization objective, and the need for external regression tracking. This experience could be captured systematically, for instance through libraries of reusable synthesis strategies or through meta-learning over past synthesis runs. Over time, the pipeline may become not only a generator of bespoke engines but a repository of database design knowledge, encoded not in a fixed codebase but in the synthesis process that produces them.

7 Conclusion and Future Work

Rethinking General-Purpose Databases. In this paper, we have argued that general-purpose database systems inherently impose a structural performance tax that cannot be fully eliminated through engineering optimizations alone. Rather than accepting this tax as inevitable, we propose an alternative: avoiding it entirely by synthesizing database engines on demand from concrete workload specifications. Our case study demonstrates that this vision is technically feasible. Starting from a workload description alone, LLM-guided synthesis produces a complete OLAP engine that executes correctly and significantly outperforms DuckDB on the target workload. These results indicate that much of the overhead in modern database systems stems not from implementation inefficiencies, but from the requirement to maintain generality.

From General-Purpose to Bespoke Engines. We do not argue that general-purpose database systems should be abandoned. Instead, we suggest they should no longer be the default choice for workloads whose structure is known in advance. For such workloads, a bespoke engine can eliminate abstractions that exist solely to preserve flexibility, replacing them with specialized data structures, fused execution paths, and workload-specific storage layouts. Today, this level of specialization is typically accessible only to organizations that can justify sustained, large-scale engineering investment. Bespoke synthesis fundamentally changes this equation: any team capable of describing its workload can obtain an engine tailored to it, independent of whether it employs database engineers. Just as compilers freed developers from writing assembly code by hand, synthesis pipelines may free data engineers from manually building database engines. The expertise does not vanish; it shifts into the pipeline, benefiting everyone who uses it.

Future Direction: Blurring Boundaries between DBMSs and Applications. The implications of this shift extend far beyond making specialization of DBMS more accessible. Once synthesis becomes an established capability, it reveals that many boundaries in current systems are artifacts of generality. For example, the interface between applications and databases is one such boundary. Today, applications submit queries and receive generic result sets, which must then be deserialized and restructured for further processing. This separation exists because the database cannot anticipate how results will be consumed. In a bespoke setting, however, both queries and their consumers are known at synthesis time.

The pipeline can generate application logic and the database engine as a single fused program, where data flows directly from storage through execution into application processing without ever crossing an API boundary. In this model, the database no longer merely returns results—the application and engine together become a single synthesized artifact.

Future Direction: True Hardware-Software Co-Design The boundary between software and hardware is equally artificial. If the workload is fully known at synthesis time, specialization need not stop at the software layer. A bespoke TPC-H machine, for instance, might take the form of a dataflow engine whose circuits implement the specific filtered scans, hash probes, and aggregation loops the workload requires, with memory controllers optimized for observed access patterns and interconnects sized for actual data movement. In such a system, the distinction between hardware and software blurs: a software hash join becomes a hardware pipeline stage, and an in-memory index becomes a dedicated lookup circuit. Historically, the complexity of co-designing hardware and software for arbitrary workloads made this approach impractical. Bespoke synthesis, however, fundamentally changes the problem: the workload is fixed, access patterns are known, and computational kernels are concrete. If AI-driven synthesis can already generate specialized database engines, it may eventually handle specialized hardware as well.

Towards an Overall Bespoke Future. In this emerging paradigm, the focus of database engineering shifts. The central question is no longer how to build a better general-purpose engine, but how to precisely specify intent and automatically translate it into an efficient, correct, and specialized system. In this vision, the engine, application interface, and potentially even hardware are outputs of a single synthesis process, fundamentally redefining what it means to engineer a database system.

References

[1] M. Stonebraker and U. Çetintemel, ""One Size Fits All": An idea whose time has come and gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 2–11.

[2] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564.

[3] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang *et al.*, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[4] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 195–206.

[5] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[6] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[8] M. Zukowski, M. Van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical dbms," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1349–1350.

[9] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 1981–1984.

[10] J. D. Greef. (2024, Jul.) Rediscovering transaction processing from history and first principles. TigerBeetle. [Online]. Available: https://tigerbeetle.com/blog/2024-07-23-rediscovering-transaction-processing-from-history-and-first-principles/

[11] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*.

[12] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[13] OpenAI, "Openai codex," OpenAI, 2025, code generation model. [Online]. Available: https://openai.com

[14] Cursor, "Cursor ide," Anysphere, 2025, aI-powered code editor. [Online]. Available: https://cursor.sh

[15] Anthropic, "Claude code," Anthropic, 2025, agentic coding assistant and developer tooling. [Online]. Available: https://www.anthropic.com/claude

[16] GitHub, "Github copilot," GitHub, 2025, aI pair programmer. [Online]. Available: https://github.com/features/copilot

[17] Transaction Processing Performance Council, "TPC Benchmark H (Decision Support) Standard Specification," https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf, 2014, version 3.0.1.

[18] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah *et al.*, "Self-driving database management systems." in *CIDR*, vol. 4, 2017, p. 1.

[19] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM international conference on management of data*, 2017, pp. 1009–1024.

[20] J. Wehrstein, B. Hilprecht, B. Olt, M. Luthra, and C. Binnig, "The case for multi-task zero-shot learning for databases," in *AIDB @ VLDB 2022*, 2022. [Online]. Available: https://drive.google.com/file/d/1nD0oacRuc180YzkIpqGM6rUJizsEic6m/view?usp=sharing

[21] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[22] T. Ziegler, P. A. Bernstein, V. Leis, and C. Binnig, "Is scalable oltp in the cloud a solved problem?" in *CIDR*, 2023.

[23] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.

[24] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "Clickhouse-lightning fast analytics for everyone," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 3731–3744, 2024.

[25] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1917–1923.

[26] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[27] M. H. Iqbal, T. R. Soomro *et al.*, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[28] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, no. 36. AISEL, 2013, pp. 141–147.

[29] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native mpp graph database," *arXiv preprint arXiv:1901.08248*, 2019.

[30] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1275–1288.

[31] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," *arXiv preprint arXiv:2201.00561*, 2022.

[32] Z. Wu, R. Marcus, Z. Liu, P. Negi, V. Nathan, P. Pfeil, G. Saxena, M. Rahman, B. Narayanaswamy, and T. Kraska, "Stage: Query execution time prediction in amazon redshift," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 280–294.

[33] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[34] T. Kraska *et al.*, "SageDB: A learned database system," in *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[35] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proceedings of the VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, 2018.

[36] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[37] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 307–322.

[38] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[39] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database system implementation*. Prentice Hall Upper Saddle River, 2000, vol. 672.

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

Non-profit Org.
U.S. Postage
PAID
Los Alamitos, CA
Permit 1398