

# Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines

Johannes Wehrstein  
Technical University of Darmstadt

Matthias Jasny\*  
Microsoft Gray-Systems-Lab

Timo Eckmann  
Technical University of Darmstadt

Carsten Binnig  
Technical University of Darmstadt & DFKI & hessian.AI

## ABSTRACT

Modern OLAP engines support arbitrary analytical workloads, but this flexibility incurs overhead from runtime schema interpretation, generic data representations, and abstraction layers, even in compiled-query systems. Workload-specific engines can eliminate these costs and exploit specialized data structures and algorithms for higher performance, yet have historically been too expensive to build manually. Recent advances in LLM-based code synthesis challenge this tradeoff, but naive prompting does not produce correct or efficient engines due to deep architectural dependencies and the need for systematic refinement. We present *Bespoke OLAP*, a fully autonomous synthesis pipeline that constructs high-performance OLAP engines tailored to a target workload through iterative performance evaluation and automated validation. *Bespoke OLAP* generates engines from scratch within minutes to hours and achieves order-of-magnitude speedups over DuckDB and Umbra, demonstrating that the generality tax extends beyond query compilation to storage layout and algorithmic design.

### PVLDB Reference Format:

Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. *Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines*. PVLDB, 19(11): 3759 - 3771, 2026. doi:10.14778/3836663.3836723

### PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/DataManagementLab/BespokeOLAP>.

## 1 INTRODUCTION

**OLAP Engines Carry an Inherent Performance Tax.** It is well established that one size does not fit all [24]. Over the past decades, this insight has led to workload-class-specific systems, including columnar OLAP engines such as DuckDB [23] and HyPer [12], which depart significantly from traditional row-store architectures. Yet even these systems follow a refined version of the “one-size-fits-all” philosophy: they are designed for *arbitrary* OLAP workloads within the relational model. They must support any schema, any valid SQL query, and a broad spectrum of access patterns. This

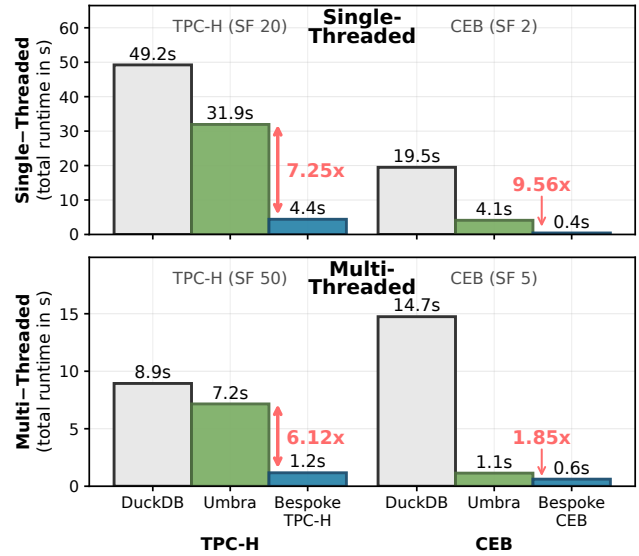
\*Work done while at Technical University of Darmstadt.

E-mail: [firstname.lastname@tu-darmstadt.de](mailto:firstname.lastname@tu-darmstadt.de)

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

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.

doi:10.14778/3836663.3836723



**Figure 1: Synthesized Bespoke engines exceed the performance of a decade of engineering, achieving 11.17× and 45.33× lower total runtime on TPC-H and CEB over DuckDB in single-threaded execution, outperforming even Umbra, which uses compiled query execution (7.24× on TPC-H and 9.56× on CEB). Similar speedups are visible in multi-threaded execution.**

flexibility introduces an inherent performance tax. Schemas are interpreted at runtime, tuple layouts remain generic, and data structures are chosen to accommodate unknown future queries. While query compilation reduces interpretation overhead for individual queries [20], engines still operate within a generic storage and execution layer where all kinds of SQL queries can be executed, and thus general-purpose algorithms and data structures for storing tables are used. Importantly, this overhead is not accidental or due to poor engineering; it is the unavoidable price of generality.

**Generality is an Overhead.** Crucially, many real-world OLAP deployments do not require such flexibility. Enterprise data warehouses frequently execute stable suites of parameterized query templates. Recent workload analyses confirm extremely high query repetition rates in practice [17, 29, 31]. In these settings, the ability to run arbitrary ad hoc queries is rarely exercised, yet its cost is incurred on every execution as a general-purpose OLAP engine is used. Instead, using a database engine designed for a specific

workload with a fixed schema and a known set of queries eliminates the overhead of generality by construction. More precisely, schema interpretation disappears entirely, storage layouts can be organized around actual access patterns, and encoding decisions can be tailored to observed value distributions. Furthermore, all queries can be compiled into code that directly operates on workload-specific data structures rather than on generic relational operators. As a consequence, entire subsystems such as dynamic planning, generic operator selection, and fallback code paths can be removed from the runtime path. The result is not a better general-purpose engine, but a different artifact: an engine correct and maximally optimized for a single workload. In short, the optimal OLAP engine for a workload is what we call a bespoke “one-size-fits-one” engine.

#### **Why Bespoke Engines Have Been Economically Out of Reach.**

If the performance benefits are so clear, why are bespoke OLAP engines not common practice? The obstacle has never been conceptual feasibility but rather economics. Building a high-performance DBMS requires years of engineering by expert teams. Storage layout, indexing, execution strategies, memory management, and testing infrastructure are deeply interdependent; even small design decisions propagate throughout the system. Constructing a bespoke engine is therefore comparable in cost to building a new database system. Rare examples such as TigerBeetle [10] demonstrate that extreme specialization can pay off when the target workload is narrow yet widely shared. However, most analytical workloads are organization-specific. In a world with potentially millions of distinct workloads, manually building an engine for each workload requires millions of engines. Engineering effort scales linearly with workload diversity, rendering a manual bespoke design economically infeasible. Consequently, as of today, general-purpose engines are used despite the clear performance benefits of bespoke engines. The performance tax of generality is therefore paid indefinitely, not because specialization does not help, but because manual specialization does not scale.

**LLM-Guided Synthesis Makes Bespoke Engines Practical.** Recent advances in AI-based code generation fundamentally change this tradeoff. Large language model (LLM) coding agents have demonstrated the ability to generate substantial, non-trivial software artifacts at low cost and within hours rather than years. This raises a natural question: can we automatically synthesize a bespoke OLAP engine for a given workload specification? In this paper, we investigate this question. Given a workload specification consisting of query templates, and performance objectives, our LLM-driven pipeline generates a complete, workload-specific analytical engine in minutes to a few hours, at a cost of only a few dollars. Importantly, the generated engine is not a configured instance of an existing system, instead, it is constructed from scratch for one workload and aggressively optimized for it. The resulting performance gains are substantial. Figure 1 shows that synthesized engines for TPC-H and CEB achieve double-digit reductions in total runtime over DuckDB and Umbra. These improvements reflect structural overhead imposed by generality rather than incremental implementation differences and demonstrate that a significant fraction of modern OLAP cost arises from remaining general-purpose design.

**The Challenge of Synthesizing OLAP Engines.** Naively prompting does not work. Initially, we tried to simply use a coding agent

to synthesize a bespoke engine. However, we ran into several problems. Firstly, correctness was not consistently reached and it took intensive human guiding and many iterations to finally generate a seemingly bespoke engine. Additionally, even if the coding agent produced a correct execution strategy it was not performant. This is mainly due to database engines being complex systems of deeply interdependent components as storage formats constrain operator design, execution strategies depend on data layout, and optimizations in one layer can introduce regressions in another. Therefore, in our early testing, full OLAP engines that are naively generated by LLMs often fail to compile, violate correctness, or exhibit fragile performance across queries. The core scientific problem is therefore not code generation per se, but structured system synthesis of complex systems, which maintains correctness while aggressively optimizing performance. As such, without principled structure and validation, LLM-driven development quickly degenerates into uncontrolled modification rather than systematic optimization.

#### **Our Contribution: A Pipeline for Bespoke Engine Synthesis.**

This paper presents a synthesis pipeline that meets this challenge. First, generation proceeds incrementally: the agent constructs and validates a functional storage layer before introducing execution logic, and queries are synthesized one by one to localize complexity and prevent error propagation. Second, correctness and performance optimization are strictly separated. We first establish a validated, correct baseline and then iteratively optimize for performance without sacrificing correctness. However, incrementally changing code and then running it to test correctness and performance comes at a high cost. To make this optimization loop practical, we thus also contribute infrastructure for system synthesis that enables live hotpatching of database engines. This allows the synthesis process to modify a running engine and immediately observe performance effects, reducing validation and benchmarking time from minutes to seconds. Candidate implementations are benchmarked on the target data (downscaled) and queries, while fine-grained runtime profiles identify bottlenecks that guide subsequent synthesis steps. Unlike classical database engines, which rely on cost models to predict optimal strategies at runtime, our approach can thus directly measure alternatives during synthesis and hard-code the fastest one into the final artifact.

**Outline.** In Section 2, we provide a more detailed overview of Bespoke OLAP synthesis. Section 3 then presents the complete design of our synthesis pipeline, followed by Section 4, which describes the infrastructure for system synthesis, which involves live hotpatching of database engines, regression tracking, and empirically driven optimization. Section 5 provides a comprehensive evaluation on TPC-H [25] and CEB [19], demonstrating consistent and substantial speedups over DuckDB across all query templates. We then discuss related work in Section 6, where we distinguish our approach from query compilation, auto-tuning, and learned components. Finally, to conclude, we summarize the contributions and outline research directions.

## **2 OVERVIEW OF BESPOKE OLAP**

In this section, we first present an overview of Bespoke OLAP. Afterwards, we also discuss the scope of this work.

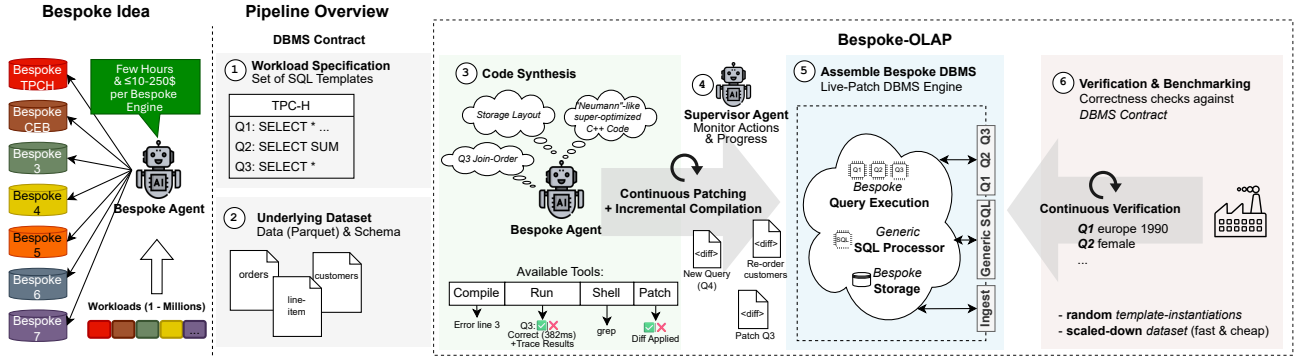


Figure 2: Overview of the Bespoke OLAP idea: one engine is generated per workload (left), and the pipeline to synthesize one engine (right). The pipeline works as follows: Based on the DBMS Contract (SQL Query Templates ① and Dataset ②), the Bespoke Agent iteratively generates the query execution engine③, checked by the supervisor agent ④, compiles, deploys⑤, and validates it⑥, and uses performance and correctness results to guide further code generation. At runtime, the generated bespoke engine exposes interfaces for ingest, for the query workload, and even ad-hoc queries, which are routed to a generic SQL processor ⑤.

## 2.1 Synthesizing Bespoke Engines

Figure 2 provides an overview of our approach. At the core is the so-called Bespoke Agent, which implements our synthesis process and generates the code for the Bespoke OLAP engines.

**A Bespoke Engine is Defined by the Workload.** The starting point of bespoke synthesis is a formal commitment of a user who wants to synthesize an engine, which we call the *DBMS contract*. This contract consists of two parts: The first part of the contract, as shown in Figure 2 at step ①, is a workload specification comprising the database schema together with a set of query templates and their parameterization ranges, which defines exactly what the engine must be capable of executing. The second part of the contract is the underlying dataset (② in Figure 2), provided as a collection of Parquet files, from which the engine derives its physical storage layout during an initial ingestion phase. Together, these two components define the complete design space available for a Bespoke engine, and every decision in step ③ in Figure 2 is grounded in what the contract reveals about the data and the queries. Naturally, a bespoke engine can only be generated for workloads whose query templates and correct query results are known in advance. Importantly, generation is done incrementally as described below, and the generated bespoke DBMS is verified and benchmarked in every iteration (see ⑥) to guarantee correctness and performance and guide the code generation.

**Bounded Scope is the Strength of Bespoke OLAP.** The contract also serves as a boundary; i.e., queries that fall outside the specification are explicitly not supported by the bespoke engine and are instead handled by a fallback database engine (i.e., a general-purpose DBMS or a generated general SQL processor over the Bespoke Storage) as shown in ⑤ in Figure 2. As a prime application of Bespoke OLAP, we see typical warehousing dashboards that execute a fixed set of queries. Additionally, any OLAP workload with repeating queries, which are highly common as discussed before, are an application for Bespoke OLAP. Importantly, the bounded scope of a Bespoke OLAP engine is not a limitation but the very source of the engine’s performance advantage, since it is precisely because the workload is fixed that the pipeline can eliminate every

abstraction that exists solely to handle cases the workload will never present. Based on the two given inputs, namely the database and query templates, the full system is generated by the Bespoke Agent, including the storage layout and data ingestion code, the data structures used to represent each table, the execution code for each query, etc.. Therefore, the engine that emerges is not a configured instance of any existing system but a purpose-built artifact whose entire internals are shaped by the contract with its workload.

**Synthesis Requires Both: Pipeline and Infrastructure.** With the contract defined and the scope fixed, the remaining question is how to reliably guide an LLM agent in constructing an optimized Bespoke engine. As we describe in detail in Section 3, this requires two things working together. (1) The first component is the synthesis pipeline (also called the agent loop in this paper) that breaks synthesis into well-defined stages, moving from storage layout planning through basic query implementation and into a multi-round optimization phase driven entirely by empirical measurement rather than cost estimation. (2) The second component is a dedicated infrastructure that keeps the development loop fast, allowing the Bespoke Agent to test the synthesized engine quickly for performance and correctness. Here, we developed a system for system generation that includes techniques for hotpatching database engines, protection against regressions through external rollback, and ensures correctness through continuous fuzzy testing. Together, these two components make reliable synthesis possible, and Section 3 and Section 4 describe both in detail.

**How to Support Workload Changes and Ad-hoc Queries?** Because a bespoke engine is specialized for a fixed workload, an important question is what happens when previously unseen queries arrive or when the workload shifts over time. We address this at two levels. For occasional ad-hoc queries that fall outside the contract, the engine can route execution through a general-purpose SQL interface that operates on the same underlying storage layer, thereby providing correct results without workload-specific optimization. For more substantial workload changes, the engine can be resynthesized entirely, and because synthesis completes in minutes to hours

at the cost of a few dollars, regeneration is a practical response to workload drift rather than a theoretical fallback and can even be done on a high frequency, such as every night during batch loading new data. Resynthesis can also either reuse the existing storage layer to avoid re-ingesting the data if the schema does not change or include a full reorganization of the physical layout to optimally support the new workload. We describe both mechanisms in more detail in Section 3.2.

## 2.2 Scope of this Work

**C++ as Language for the Engine** Our pipeline is language agnostic. However, an early design decision we made in the pipeline for our experiments is the target programming language, as it determines the quality of compiler feedback, the available optimization headroom, and how reliably agents can reason about generated code. We chose C++ because it is well understood by current LLMs, enabling idiomatic and performant code generation. It provides fine-grained control over memory layout and low-level execution, where workload-specific optimizations arise; higher-level languages would obscure these physical decisions. Additionally, its tooling and compilation infrastructure offers clear feedback to the LLM.

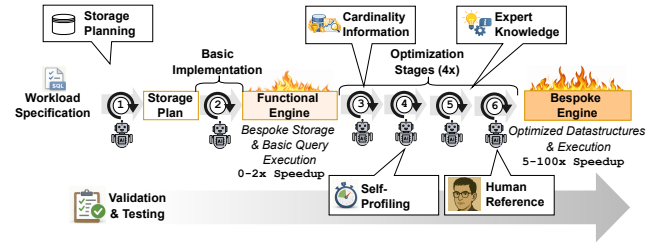
**In-Memory OLAP Engines.** To show that bespoke system generation is possible and see the order in which we can expect benefits, the synthesis pipeline described in this paper limits the design to in-memory engines. This eliminates the need to generate code for buffer management, page eviction, and I/O scheduling from the design space, allowing the agent to focus its optimization effort on data structure design and computation rather than on data movement between storage tiers. We evaluate both single-threaded and multi-threaded execution. Single-threaded results isolate the contribution of specialization from parallelism, and multi-threaded results demonstrate that the synthesized engines scale effectively with additional cores. These are deliberate restrictions rather than fundamental limitations of the bespoke synthesis approach to see how well bespoke adaptation on the full system level can help. Within this scope, we demonstrate that bespoke synthesis can produce engines that substantially outperform highly optimized general-purpose systems, which is the central claim this paper sets out to establish. Extending the pipeline to disk-resident data introduces new challenges that we mention as future work in Section 7. All generated engines build on the same core synthesis methodology, which we describe in detail next.

## 3 THE GENERATION PIPELINE

In this section, we describe the generation pipeline for synthesizing bespoke OLAP engines, walking through each stage from storage planning through basic query implementation to the four steps of optimization that produce the final bespoke engine. An overview of all the synthesis stages is illustrated in Figure 3.

### 3.1 Overview of Pipeline

**The Environment of the Bespoke Agent.** Throughout the entire synthesis process, the Bespoke Agent interacts with the environment exclusively through four tools shown in Figure 2 (3). The first is a *compiler tool* that compiles the current state of the codebase



**Figure 3: Steps to produce a Bespoke Engine.** Each step is executed in loops, with validation to check whether the step’s goal is achieved. ① produces a storage plan for the specified workload. ② produces a functional engine based on the storage plan coupled with continuous validation and testing to steer the iterative coding process (see arrow below). The optimization loop produces an optimized engine iteratively by leveraging additional information such as actual summaries of the data ③, self-profiling ④, expert knowledge from the DBMS research literature ⑤, and taking the perspective of expert database engineers ⑥.

and returns any errors with precise file and line information, giving the agent actionable feedback whenever a change introduces a syntactic or type error. The second is a *benchmark tool* that executes the current engine against the workload, reports correctness by comparing results against DuckDB across a set of parameter instantiations, and returns runtime measurements for each query. The third is a *shell tool* that allows the agent to inspect the underlying dataset, examine intermediate values during execution, and explore the structure of the codebase. The fourth is a *patch tool* through which the agent submits all code changes as structured patches rather than full file rewrites. This last tool is the interface through which hotpatching operates, which we explain more in Section 4. All four tools together define the complete boundary between the agent and the system it is building, and the agent has no mechanism to interact with the engine other than through them.

**Synthesis Begins With Storage Layout Planning.** The first stage of the agent loop is storage layout planning, shown in Figure 3 ①, and it is deliberately the only stage in which no execution code is written. Before the agent generates a single query function, it uses the shell tool to examine the underlying dataset in detail and cross-references its findings with the query templates in the workload specification. It identifies which columns appear in predicates, which attributes serve as join keys between tables, which columns are accessed together by multiple queries, and how data is distributed across the relevant attributes. Based on this analysis, the agent decides on the physical storage layout for each table (called storage plan), selecting sort orders, encoding schemes, and auxiliary data structures that are optimal for the specific access patterns the workload presents. This planning phase is completed and committed before any code is written. The reason for this strict ordering is that storage decisions affect every query in the workload simultaneously. Allowing the agent to revisit storage decisions during query implementation would introduce cross-query dependencies that are very difficult to manage reliably and would undermine the correctness of already-validated queries. Hence, changes to the storage layout should be kept to a minimum to avoid this overhead.

Once the initial storage layout is determined, the agent proceeds to the second stage and generates the corresponding C++ structs ② for storing the data together with the ingestion code that loads data, transforms them according to the storage layout, and populates the in-memory storage representation. After ingestion, the code is compiled, and the database process is started, which then remains running until the complete synthesis finishes.

**The Base Ingredients: Correctness & Tracing.** With a running engine and a stable storage layer in place, the Bespoke Agent next proceeds to implement a dedicated execution function for each SQL query template in the workload specification. At this stage, the goal is to generate an engine that correctly executes all queries, rather than optimizing for performance. Correctness is checked by comparing query results against an existing engine such as DuckDB. This basic implementation phase establishes the known-good baseline that all subsequent optimization will be measured against. Once the full set of query templates executes correctly, the agent performs two preparatory steps before optimization begins. The first is the addition of CPU core pinning, for which we provide C++ helper functions at specific positions in the execution code, to ensure reliable benchmarking results. The second preparatory step is the installation of tracing functionality, which includes lightweight performance counters, loop iteration trackers, and timestamps that the agent autonomously injects into the execution code wherever it finds additional statistics meaningful. These agent-injected tracing steps will later be used by the agent to verify its implementation decisions and optimize the query code for performance. All tracing code is guarded by a preprocessor directive and can be disabled by the agent, such that it imposes no overhead for the final engine.

**Optimization Proceeds in a Supervised Loop.** With the instrumentation in place, the Bespoke Agent enters the optimization phase, which proceeds through four rounds of increasing sophistication (shown in the middle of Figure 3 ③-⑥). A dedicated supervisor agent observes each turn of the optimization loop, verifying that the agent invoked the appropriate tools, completed the assigned task, and introduced no regressions. When the supervisor detects gaps or errors, it injects targeted corrective feedback directly into the agent’s context; otherwise it approves the turn and the loop advances. The supervisor is stage-aware: it knows the sequence of upcoming stages and refrains from pushing the agent to address concerns that are reserved for a later round. In the first round ③, the agent receives signals from executing one randomly sampled query plan per query template from DuckDB, annotated with automatically retrieved actual cardinalities. This additional information about cardinalities for one possible instantiation of a query template is the starting point for the agent to iteratively try and empirically evaluate optimization decisions, such as join order and operator implementations, as we will discuss later. Bootstrapping this empirical optimization loop with the cardinalities of one sample instantiation per query-template allows the agent to skip highly suboptimal join-orders from the search space right from the start, speeding up the optimization process.

In the second round ④, the agent activates the tracing framework ④ and analyzes the instrumentation output it produces. It examines where time is spent within each query function, identifies loop bodies that dominate execution time, and pinpoints operators whose implementation does not match the scale of intermediate

results. This self-directed profiling allows the agent to localize bottlenecks precisely rather than making speculative changes to the entire function. In the third round ⑤, the agent draws on distilled expert knowledge ⑤ from the database research literature that we have curated and provided as part of the synthesis environment. This knowledge covers established optimization techniques from decades of systems research, focused on keeping data access sequential, reducing cache misses and copies, hoisting loop invariants, minimizing branches, and removing expensive work from inner loops. It also recommends choosing data-appropriate algorithms and containers, pre-sizing outputs, using direct parsing and constant-time lookups, and helping the compiler and hardware with aliasing hints, inlining, and SIMD-friendly code<sup>1</sup>. The agent applies these techniques selectively to the bottlenecks it identified through tracing and validates each application against the benchmark tool to confirm that the expected improvement materialized. In the fourth and final round ⑥, the agent adopts the perspective of an expert database engineer and performs a holistic review of each query function, looking for any remaining inefficiency that the previous rounds did not address. It treats the tracing output as its primary guide and reasons about the generated code the way a systems engineer would, questioning every allocation, every branch, and every data movement that cannot be justified by the workload’s access patterns. Changes made in this round are validated and checkpointed exactly as in earlier rounds, so that any regression is detected immediately and reverted before it can affect other queries.

**No cost modeling, but empirically finding join-orders.** Since query templates are parameterized, different instantiations can yield vastly different intermediate cardinalities and thus different optimal join orders, something a static cost model in a generic DBMS would have to predict. In practice, however, accurate cost models for arbitrary operator implementations do not exist. Our Bespoke Agent therefore sidesteps cost modeling entirely and discovers effective join orders empirically: it proposes an ordering, benchmarks it on sampled instantiations, and iteratively refines the design based on observed runtimes. The cardinality information with an optimal plan, as provided in Figure 3 ③, eliminates suboptimal join orders early, narrowing the search space before deeper exploration. When parameter instantiations consistently favor different join orders, the agent either (1) selects a single robust plan that performs well across samples or (2) generates separate specialized implementations for distinct regions of the parameter space, tailoring the code to specific argument profiles. This iterative propose-and-evaluate loop enables the bespoke agent to accommodate the full diversity of a query template’s parameter space. **Multi-Threaded Execution as a Fifth Optimization Stage.** Once single-threaded optimization is complete, the pipeline proceeds to a fifth stage that extends the synthesized engine to parallel execution. The agent first adds a shared ThreadPool that all query functions access. This pool is initialized from a CORE\_IDS environment variable, which determines both thread count and CPU core pinning. It then parallelizes each query independently using the same conversation-branching infrastructure as before. For each query, the agent consults the single-threaded tracing output to identify the dominant bottleneck

<sup>1</sup>The complete expert-knowledge file can be found [here](#).

and then selects an appropriate decomposition strategy, such as range-based scan partitioning, per-thread local aggregation with a sequential merge, or parallel hash joins. In doing so, it targets the operator that accounts for the largest share of runtime rather than parallelizing indiscriminately. Correctness and regression tracking apply throughout, exactly as in the single-threaded stages.

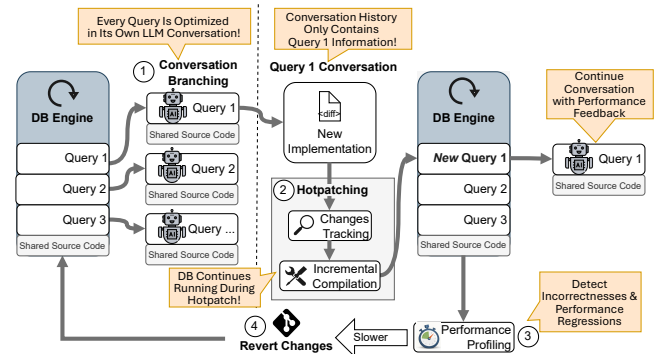
### 3.2 Resynthesizing with Changing Workload

A bespoke OLAP engine is, by design, specialized for a fixed workload, and this naturally raises the question of what happens when the workload changes. Two distinct scenarios must be addressed, and each calls for a different response.

**Ad-hoc queries outside the workload contract.** The first scenario concerns occasional ad-hoc queries that fall outside the workload contract. A central invariant of the synthesis pipeline is that all data must remain materializable into a flat relational format at all times, regardless of how aggressively the bespoke engine compresses, reorders, or reorganizes it internally. The pipeline enforces this invariant during synthesis, which guarantees that the original relational semantics of the data are never lost, even when the internal representation deviates substantially from a conventional table layout. Because this invariant holds, the materialized flat representation can be handed to any general-purpose OLAP engine, such as DuckDB, which then serves as a fallback for queries that the bespoke engine was not designed to handle. The performance of this fallback path is naturally comparable to what the user would have obtained from a conventional system in the first place, so the worst case for an out-of-contract query is not failure but rather the absence of workload-specific optimization. This means that deploying a bespoke engine never makes the system less capable than a general-purpose alternative; it only adds a fast path for the queries that the contract covers.

**Sustained workload drift.** The second scenario concerns sustained workload drift, in which the set of queries the system is expected to execute changes substantially over time. In this case, the appropriate response is to resynthesize the engine entirely for the new workload. The key observation that makes this practical is that resynthesis is both fast and inexpensive. Because a full engine can be generated in the order of minutes to hours at the cost of a few dollars, resynthesis is not a catastrophic recovery procedure but can be a routine maintenance operation similar to data loading. This stands in sharp contrast to manually engineered specialization, where adapting to a new workload requires weeks or months of tuning effort and is therefore avoided unless absolutely necessary.

**Modes of resynthesis.** Furthermore, resynthesis offers two modes of operation depending on the extent of the change. If the new workload operates on the same schema and data but requires different query implementations or execution strategies, the engine can be resynthesized while keeping the existing storage layer intact, avoiding the cost of reingesting the data entirely and reducing deployment time. If the new workload also demands a fundamentally different physical layout, for instance, because new queries access different column combinations or require different sort orders, a full resynthesis including storage reorganization can be performed instead. In both cases, the hotpatching infrastructure described in Section 4 allows the updated engine to be deployed without



**Figure 4: Overview of the infrastructure supporting LLM-driven OLAP engine synthesis.** The Bespoke Agent first performs shared preparation and then branches into independent per-query optimization threads (①). Code modifications are applied through hotpatching (②), allowing incremental recompilation and replacement of individual components without restarting the running database process. Each change is automatically validated using fuzzy testing and benchmarking against DuckDB to ensure correctness and measure performance (③). All states are tracked using snapshot-based versioning with external regression monitoring (④), enabling rollback, reproducibility, and monotonic optimization. This infrastructure provides a fast, correct, and recoverable synthesis loop.

restarting the running process, so that the transition from the old workload to the new one happens without downtime.

## 4 A SYSTEM FOR SYSTEM GENERATION

As shown in the previous section, synthesis of a Bespoke OLAP engine is not a single step but an incremental process that moves from a workload specification to a fully optimized engine through a sequence of well-defined stages. To make this process work reliably and at a reasonable cost and time, it requires more than a capable LLM. It requires an infrastructure that keeps the development loop fast, protects against mistakes, and ensures that every change the Bespoke Agent makes is immediately verifiable. This section describes both the infrastructure we built to support synthesis and the Bespoke Agent loop that uses it. We begin by describing the system we built for system generation, covering the mechanisms that make synthesis fast, correct, and recoverable.

**System Generation Requires a Dedicated Infrastructure.** Generating a database engine with an LLM coding agent is fundamentally different from asking an LLM to write a standalone function or a small script. A database engine is a living system that must be compiled, loaded with data, exercised under queries, measured for correctness, and profiled for performance, and all of this must happen repeatedly and quickly during synthesis. Without infrastructure designed specifically to support this loop, each iteration of the Bespoke Agent’s development cycle becomes prohibitively slow. The Bespoke Agent would need to recompile the entire codebase on every change, reload data from scratch after every modification, and wait for full system restarts before observing the effect of its edits. At that pace, synthesis of a complete OLAP engine would

take far too long and be impractical. With the infrastructure we propose in the following sections, shown in Figure 4 and Figure 2, synthesis can be completed in minutes to hours rather than days.

#### **Conversation Branching to Enable Per-Query Optimization.**

Another insight into synthesizing a system for a workload is the importance of managing the Bespoke Agent’s conversation history (the context the agent operates within) throughout incremental generation. Overall, managing this context is as important as managing the code it produces. As discussed before, during the system generation phase, at the beginning, some steps are shared across all queries. We therefore begin the generation with a shared conversation for all queries in which the Bespoke Agent performs global preparation, including planning storage and generating the storage code. Inconsistency in the storage layer, the query interface, and how pinning and tracing components are used across queries would prevent any deep optimizations, and make the output result unreliable and performance unstable. The shared foundation consists of three concrete artifacts: an implementation plan specifying the storage approach and data structures, the builder code (`builder.cpp`) that ingests Arrow/Parquet tables into the in-memory layout, and query interface stubs with agreed-upon function signatures. The foundation is complete once the builder compiles and the ingest path is validated against a reference implementation. Once the shared foundation is established, however, we break the conversation into a separate thread per query (called conversation branching). This can be seen at step ① of Figure 4 where we split the context per query.

For the branching conversation, we split the synthesis into multiple Bespoke Agents, one agent thread per query template. Each per-query agent thread carries the shared history as its starting point, but then develops its context independently, focusing entirely on optimizing a single query without being distracted by the implementation details of others. This is possible because queries can be optimized independently without risk of interference. However, the shared components, such as storage, can still be updated by individual agents to improve storage based on insights gained when synthesizing code for a single query. To enable this and still have one shared storage for all queries, each agent contributes their changes to the shared sequence so that changes to shared components remain consistent. Concretely, the first per-query Bespoke Agent commits its changes before the second begins, and so on through the sequence. Importantly, the order in which queries are processed does not affect correctness or performance, as multiple rounds of implementations, evaluations and optimizations are applied. Because each branch operates live on the same running database engine, each per-query Bespoke Agent can observe any changes by a preceding Bespoke Agent, including both general changes on the engine and interfaces, as well as the storage layout, and can therefore build correctly on top of them. The result is that each Bespoke Agent reasons within a compact and focused context while still operating on a globally consistent codebase.

**Fast Iteration through Continuous and Restart-Free Deployment.** The most consequential infrastructure component is the hotpatching engine, which can be seen in step ② of Figure 4. Rather than treating the synthesized database as a static binary that must be fully recompiled and restarted after every change, our system

keeps the database process running continuously throughout synthesis. When the Bespoke Agent modifies a specific component of the engine, such as the execution function for a single query or a helper data structure, the system recompiles only that component in isolation and swaps it into the running process without interruption. The consequence of this design is significant. Because the process never stops, the storage layer remains resident in memory at all times. This means the Bespoke Agent never needs to re-run the ingestion pipeline after a code change, which at a large scale factor of data would otherwise consume minutes per iteration. A data reload is only necessary when the Bespoke Agent decides to modify the storage layer. Combined with incremental recompilation, hotpatching reduces the turnaround time between a Bespoke Agent edit and an observable result from minutes to seconds. This reduction is not merely a convenience. With hundreds of patching steps required to produce a full, highly optimized TPC-H engine, this expansion increases the number of optimization candidates the Bespoke Agent can explore within a fixed time budget and therefore affects the quality of the final synthesized engine.

**Correctness needs Continuous & Fast Testing.** Correctness validation in our pipeline (see ③ in Figure 4) goes beyond checking that a query returns the right result for a single fixed input. Because query templates are parameterized, a query implementation may be correct for one parameter instantiation and subtly wrong for another. To catch such cases, the benchmark tool for the Bespoke Agent runs each query template across a diverse set of parameter instantiations within the allowed ranges defined in the workload specification. This testing approach significantly increases the probability of detecting implementation errors that a single fixed test case would miss. The results are compared against DuckDB for every instantiation. To speed up validation time and ensure that correctness checks are cheap, validation runs on a smaller scale factor during the implementation phase to keep turnaround fast. For this, using the dataset scaler of [30], testing is repeated on several scaled-down versions of the dataset to ensure consistency with slightly different data, preventing hardcoding to exact row-Ids or counts. We only switch to the full dataset size during the performance optimization phase, where runtime measurements must reflect production conditions. However, if computational cost and synthesis time are a preference, the full pipeline could run on scaled-down versions of the datasets and extrapolate performance improvements to the full dataset.

**Importance of Regression Tracking & Rollbacks.** LLM coding agents are effective at generating code changes but are systematically unreliable at recognizing when a change has made things worse. A Bespoke Agent that has just introduced a regression will often attempt to fix it with further changes, compounding the problem rather than reverting it. To prevent this, our system tracks the performance impact of every change the Bespoke Agent makes through an external monitor that is entirely outside the Bespoke Agent’s control. After each change, the benchmark tool reports the updated runtime for all affected queries, and the external monitor compares this against the most recent correct code base, which we call the *accepted baseline*. If a regression is detected, the monitor immediately rolls back the Bespoke Agent’s changes to the last known-good state and signals the Bespoke Agent to explore a different direction. This ensures that in case the Bespoke-Agent

fails to recognize its own degradation, it cannot accidentally continue on a regressed state and potentially get stuck in a suboptimal direction. This mechanism ensures that the synthesis process is monotonically improving and that the Bespoke Agent’s exploration of the optimization space never undoes progress that has already been validated.

**Code Versioning Enables Reproducibility and Recovery.** Every incremental change the Bespoke Agent makes to the codebase is tracked by a high-performance snapshotting system that we have built on top of Git (see ④ of Figure 4). This versioning layer serves three purposes. First, it is the foundation on which the roll-back process operates, since every snapshot represents a state to which the monitor can return if a regression is detected. Second, it enables full reproducibility of the synthesis process, because the complete sequence of code states from initial storage layout through final optimized engine can be replayed at any point. This is particularly valuable when a human engineer wishes to inspect or modify a specific intermediate state or to adjust a prompt and observe the downstream effect without re-running the entire synthesis from scratch. Third, versioning provides a durable backup of all intermediate states, which protects against infrastructure failures that would otherwise require synthesis to restart from the beginning. Together, these properties make the synthesis process not just fast and reliable but also transparent and recoverable at every stage. **Lessons Learned: Infrastructure as the Enabler of LLM-Driven Synthesis.** Taken together, these infrastructure components reflect a central lesson from building Bespoke OLAP: LLMs perform substantially better when presented with specific and bounded subproblems. Hotpatching, conversation branching, context compaction, fuzzy correctness testing, and external regression monitoring all exist for this reason. Each component reduces the complexity of the problem the agent faces at any given moment, ensuring it always operates on a concrete and verifiable task rather than an underspecified global objective. The value of the pipeline is therefore not in making the LLM smarter, but in structuring the synthesis task so that the LLM’s strengths are exploited and its failure modes are contained.

## 5 EXPERIMENTAL EVALUATION

In this section, we evaluate the impact of synthesizing bespoke OLAP engines. We demonstrate our approach on two established workloads from benchmarks: TPC-H and CEB, and organize our evaluation around three central questions: (1) What speedups does Bespoke-OLAP achieve over a state-of-the-art general-purpose OLAP engine? (2) Where do these speedups originate - what are the individual contributions of the optimization stages, the bespoke storage design, and the specific strategies the agent employs? (3) How does the synthesis process itself behave in terms of development effort, tool usage, and correctness?

### 5.1 Experiment Setup

**Benchmarking Workloads & Hardware.** We employ two complementary benchmarks to evaluate Bespoke-OLAP. TPC-H is a leading industrial benchmark for OLAP systems that features well-defined analytical queries for which existing engines are heavily optimized. To complement TPC-H, we additionally chose CEB [19],

a templated version of the JOB benchmark [16] that operates on the real-world IMDB dataset. For both benchmarks, we follow established data generation and instantiation procedures to ensure reproducibility. TPC-H data was generated according to the TPC-H v3.0.1 specification [25] using the DuckDB data generator. For CEB, we sampled query instantiations from the published artifacts of [19] and used the data downscaler and upscaler provided by [30] to obtain different scale factors. All experiments were executed on an AMD EPYC 9654P 96-Core Processor with 768 GB RAM.

**Bespoke Engines and Baseline.** We ran our Bespoke-OLAP agent on both benchmarks separately, producing two distinct bespoke engines: Bespoke-TPCH and Bespoke-CEB. Both implementations were created using the same framework with identical tools, prompts, and configurations. No manual interventions were performed at any stage. As a baseline, we use DuckDB (version 1.4.1), a state-of-the-art embedded analytical database engine. We additionally compare against Umbra (version 26.02), a state-of-the-art compiled-query engine that already eliminates query-interpretation overhead. To ensure a controlled comparison, DuckDB, Bespoke-OLAP and Umbra operate in single-threaded mode, pinned to the same core, in all experiments except the multi-threaded scaling study in Section 5.2.3. Furthermore, to guarantee all data resides in main memory, and DuckDB/Umbra has no I/O disadvantage, we loaded data from Parquet files into DuckDB/Umbra tables within an in-memory session before each experiment.

**LLM Models and Configuration.** All experiments in this paper use Claude Sonnet 4.7 as the primary model. We selected it for its speed, cost-efficiency, and reliable tool use across long synthesis runs. However, the pipeline is model-agnostic by design, and we have tested it with a range of other models. Claude Sonnet 4.7 and GPT 5.4-Codex achieved comparable speedups. We additionally tested locally-hosted open-weight models, including Gemma-4-26B-A4B-it, MiniMax-M2.7, GLM-4.7-FP8, and GLM-5.1-FP8, which generally achieved lower speedups and required more attempts per stage. We attribute this to model capability and instruction-following quality rather than a fundamental limitation of the approach.

### 5.2 Performance and Scalability

**5.2.1 Overall and Per-Query Speedups.** Figure 1 presents the overall speedup achieved by Bespoke-OLAP. On TPC-H at scale factor 20, Bespoke-TPCH completes the full workload in 4.4 s compared to 49.2 s for DuckDB, yielding an overall speedup of **11.17×**. On CEB at scale factor 2, Bespoke-CEB finishes in 0.4 s versus 19.5 s for DuckDB, corresponding to a speedup of **45.33×**.<sup>2</sup> Crucially, these speedups are not driven by a few outlier queries. As Figure 5 reveals, Bespoke-OLAP outperforms DuckDB on *every single query*, with per-query speedups ranging from 2.83× to 102× on TPC-H and from 11.6× to 1500× on CEB. The consistently high speedups across all queries underscore that Bespoke-OLAP’s advantages are systematic rather than accidental. In comparison to Umbra, Bespoke-OLAP achieves overall speedups of **7.24×** on TPC-H and **9.56×** on CEB over Umbra (single threaded execution). Remarkably,

<sup>2</sup>Query 8a of CEB has been excluded from all plots. DuckDB exhibited vastly spiking runtimes depending on the query instantiation, ranging from 10× to over 100×. Due to these highly unstable results, Q8a was excluded. It is worth noting, however, that Bespoke-CEB maintains very stable runtimes across all instantiations of this query.

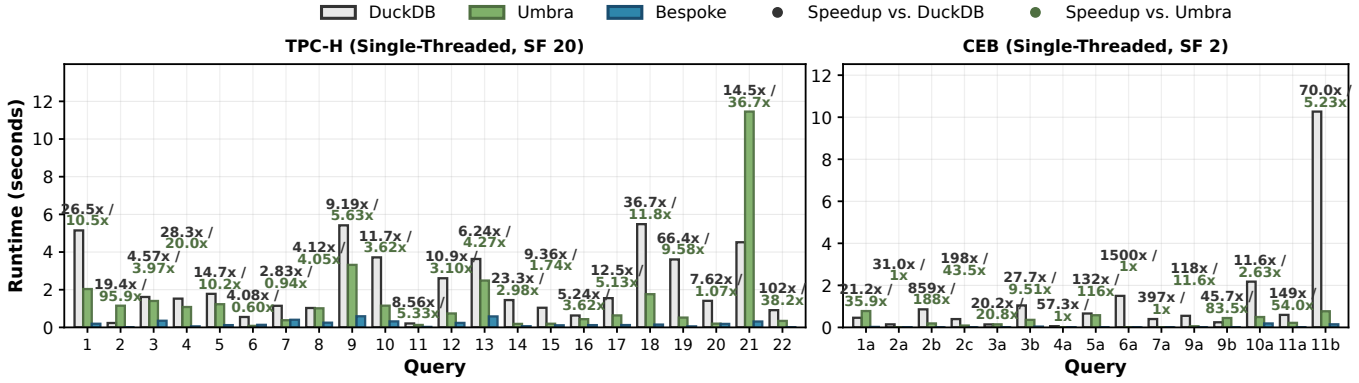


Figure 5: Per-query runtimes comparing Bespoke-TPCH/CEB and Umbra on TPC-H (SF20) and CEB (SF2) single-threaded.

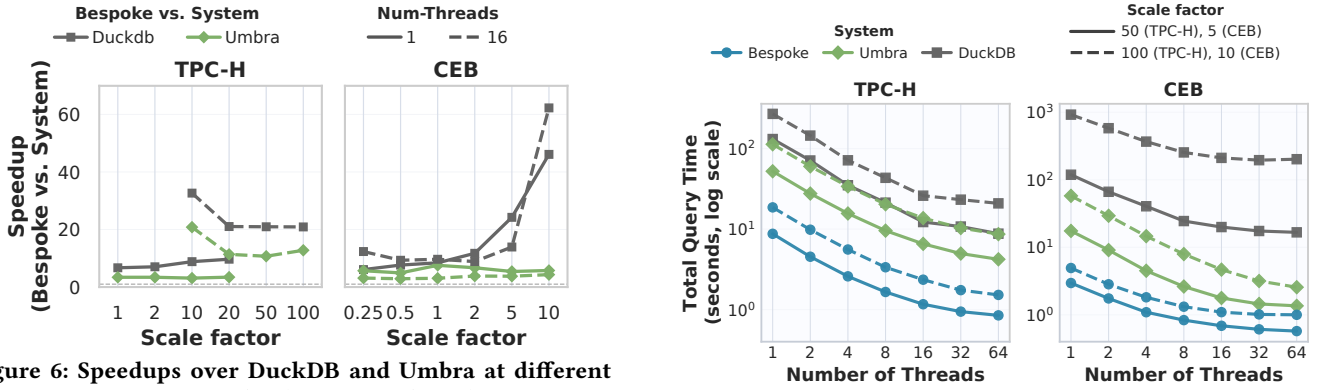


Figure 6: Speedups over DuckDB and Umbra at different scale factors for TPC-H (left) and CEB (right). On TPC-H, the speedup remains stable. On CEB, Bespoke-OLAP scales increasingly better than DuckDB, with the speedup growing to over 45 $\times$  at the largest scale factor. This stems from an algorithmically lower complexity class in Bespoke-CEB compared to DuckDB, producing increasing speedups with larger data sizes.

Bespoke-OLAP outperforms Umbra on all but two queries across both benchmarks combined, with the two exceptions at 0.60 $\times$  and 0.94 $\times$ . This demonstrates that the generality tax extends beyond query compilation into storage layout and algorithmic design: even a compiled-query engine that eliminates interpretation overhead cannot match a workload-specific engine that also optimizes storage organization and query algorithms for the given workload. The high-speedup gains on TPC-H also demonstrate the overall efficiency of Bespoke engines, as OLAP engines are already highly tuned for TPC-H and sometimes even implement specific code paths to execute such queries optimally, achieving good results on the TPC-H benchmark leaderboard.

**5.2.2 Scalability with Increasing Data Sizes.** We now examine if the performance advantages persist as dataset sizes grow. Figure 6 shows that the synthesized engines scale gracefully. On TPC-H, the speedup remains stable across all tested scale factors, indicating that Bespoke-TPCH and DuckDB exhibit the same asymptotic complexity on this workload. On CEB, however, Bespoke-OLAP scales *significantly better* than DuckDB: the speedup grows substantially with increasing scale factors, reaching over 45 $\times$  at scale

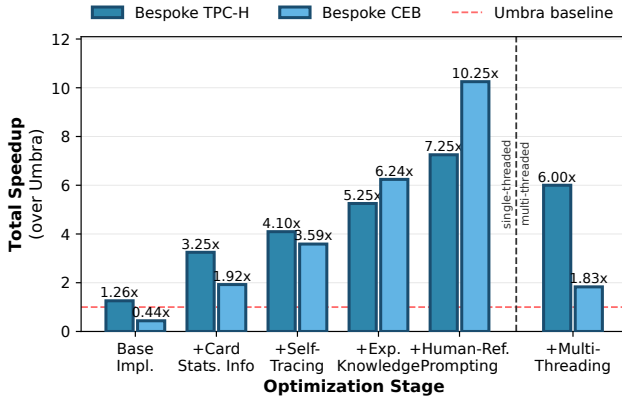
Figure 7: Multi-threaded scaling on TPC-H and CEB. Bespoke remains the fastest system at every thread count. In multi-threaded execution, Bespoke achieves 7.65 $\times$  over DuckDB and 6.12 $\times$  over Umbra on TPC-H. On CEB the speedups are 23.97 $\times$  over DuckDB and 1.87 $\times$  over Umbra.

factor 10, as DuckDB’s runtime grows disproportionately. This widening gap on CEB can be attributed to Bespoke-CEB’s carefully tuned storage layouts and query implementations, which are more robust to the increased data volumes and the complex, correlated data distributions present in the IMDB dataset. Together, these results demonstrate that Bespoke-OLAP not only delivers substantial speedups at a fixed scale, but that these advantages are maintained - and in some cases amplified - as the data grows.

**5.2.3 Scalability with Increasing Number of Threads.** Figure 7 shows total query time as the number of threads increases from 1 to 64 on both benchmarks. Bespoke remains the fastest system at every thread count on both benchmarks. In multi-threaded execution, Bespoke achieves 7.65 $\times$  over DuckDB and 6.12 $\times$  over Umbra on TPC-H. On CEB the speedups are 23.97 $\times$  over DuckDB and 1.87 $\times$  over Umbra.

### 5.3 Anatomy of the Speedups

The previous experiment established *what* Bespoke-OLAP achieves; we now turn to understanding *where* these speedups originate. We decompose the performance gains along three dimensions: the incremental impact of each optimization stage, the contribution



**Figure 8: Cumulative impact of each optimization stage on the overall speedup for TPC-H and CEB. Each bar includes all optimizations from preceding stages.**

of bespoke storage layouts versus query-execution code, and the concrete strategies the agent employs at the code level.

**5.3.1 Impact of Optimization Stages.** As described in 3, the pipeline executes four consecutive optimization stages. Figure 8 shows the measured speedup after each stage, where each stage includes all optimizations from preceding stages. Importantly, we can see that every optimization stage contributes meaningfully, confirming the value of the multi-stage pipeline design.

On TPC-H, the initial implementation already achieves a speedup of 2.34× over DuckDB, even though the generated query-execution code is entirely unoptimized. This initial gain stems solely from the bespoke storage design, which avoids the overhead of a general-purpose storage layer such as schema interpretation. To give some examples of what drives this: `lineitem` is sorted primarily by `l_shipdate` and secondarily by `l_orderkey`, enabling zone-map pruning that skips entire 64K-row blocks for the date-range filters present in fourteen of the twenty-two TPC-H queries. The four most frequently aggregated columns are packed into an 8-byte AoS struct so that a single cache-line fetch retrieves all four values per row. Furthermore, a pre-computed derived column stores the discounted price, eliminating a per-row multiply in the innermost loop of thirteen queries. The full set of storage decisions is documented in this artifact.<sup>3</sup>

On CEB, by contrast, the initial implementation achieves only 2.10× over DuckDB and remains at a *slowdown* relative to Umbra (0.44×). Here, the more complex access patterns arising from the real-world IMDB dataset, with its irregular data distributions and many-to-many relationships, require more advanced processing strategies that the initial execution code cannot yet provide; the bespoke storage alone is insufficient to compensate. Subsequent optimization stages progressively improve performance. With additional actual cardinality information, self-tracing of the generated implementation, and the injection of expert knowledge, each stage contributes measurable speedups on both benchmarks. Notably, the final stage, in which we provided the agent with a human reference persona for the code it was writing and the optimizations it was performing, yielded a substantial additional improvement.

<sup>3</sup>The storage plan can be found here.

|       | w/ Flat Storage |                   | w/ Bespoke Storage |                   |
|-------|-----------------|-------------------|--------------------|-------------------|
|       | Basic Impl.     | After Optim. Loop | Basic Impl.        | After Optim. Loop |
| TPC-H | 1.26            | 5.18              | 2.34               | 12.35             |
| CEB   | 0.57            | 8.09              | 2.10               | 51.40             |

**Table 1: Speedups over DuckDB with flat storage (struct-of-arrays) vs. bespoke storage. Flat storage shows the benefits of bespoke query code only; bespoke storage adds workload-specific physical data layouts.**

To give a concrete example: for Query 1, the prior stages had produced a sequential scan of `l_shipdate` with 4-way loop unrolling and branch-prediction hints. The human persona stage instead restructured the storage builder to partition the `lineitem` rows into nine groups by (`returnflag`, `linestatus`) key, sorting each group by `shipdate` at ingest time. This allowed the query to replace the full scan with a binary search per group, followed by a pure sequential AVX-512 reduction processing 16 elements per cycle with two independent accumulator vectors for instruction-level parallelism. The result is near-peak memory bandwidth utilization. The final Bespoke-TPC-H engine is approximately 11,600 lines of C++. This step led the model to exploit techniques it had not previously applied, such as more aggressive low-level optimizations and tighter data-access patterns, resulting in final speedups of 12.35× on TPC-H and 51.40× on CEB over DuckDB.

**5.3.2 Impact of Bespoke Storage.** The optimization ablation demonstrates that each stage contributes, but it does not disentangle the effects of bespoke *storage* from bespoke *query execution*. To isolate these two dimensions, we ran the Bespoke-OLAP agent when fixing storage to a flat design, a simple struct-of-arrays representation with no workload-specific tuning and compared it to the full bespoke generation with bespoke storage.

Table 1 shows the results, which reveal two important findings. First, even with a flat storage layout, the Bespoke-OLAP agent achieves speedups of 1.26× on TPC-H and approaches parity on CEB (0.57×) in the basic implementation phase alone. These gains, modest as they are, arise purely from removing the layers of abstraction inherent in a general-purpose query engine: the generic processing model, the iterator framework, and the type-dispatching overhead. This confirms that *the agent can generate efficient query-processing code* independent of the storage design. Second, when enabling bespoke storage, the speedups increase dramatically to 12.35× on TPC-H and 51.40× on CEB. The bespoke storage component thus amplifies the gains from efficient query execution by providing physical data layouts tailored to each query’s access patterns. Taken together, these results demonstrate that Bespoke-OLAP derives its performance from a synergy of two capabilities: generating efficient, abstraction-free query-processing code *and* designing highly beneficial storage layouts.

**5.3.3 Analysis of Employed Strategies.** To gain deeper insight into *how* the Bespoke Agent achieves these speedups, we systematically classified the strategies employed across the storage, query-execution, and low-level optimization layers in both workloads.

Table 2 presents a comprehensive summary of the bespoke and non-bespoke techniques found in the generated code. The results reveal a rich repertoire of strategies spanning six categories: column encoding, query-support structures, scan and filter strategies, join

| Bespoke and Non-Bespoke Database Strategies |                                               | Classical /Exotic | used in % of   |
|---------------------------------------------|-----------------------------------------------|-------------------|----------------|
| Column Encoding Strategies                  | Dictionary Encoding                           | C                 | 25.00%/cols    |
|                                             | Physical Row Sort Order                       | C                 | 18.75%/cols    |
|                                             | Flat String Arena (offset array)              | C                 | 10.94%/cols    |
|                                             | Scaled Integer / Fixed-Point                  | E                 | 7.03%/cols     |
|                                             | Compact Date (16-bit offset)                  | E                 | 1.56%/cols     |
| Query-Support Structures                    | Hash Index (O(1) join lookup)                 | E                 | 47.37%/queries |
|                                             | Precomputed Derived Column                    | E                 | 36.84%/queries |
|                                             | Sorted Range Directory (join key)             | C                 | 26.32%/queries |
|                                             | Precomp. Arith. Lookup Table                  | E                 | 2.63%/queries  |
| Scan & Filter Strategies                    | Dictionary Predicate Rewrite                  | E                 | 71.05%/queries |
|                                             | Sorted Key Range Lookup                       | C                 | 18.42%/queries |
|                                             | Shard-Skip Scan (Zone Maps)                   | C                 | 13.16%/queries |
|                                             | Sorted Range Scan                             | C                 | 13.16%/queries |
| Join Operators                              | Bitmap Semi-Join                              | E                 | 73.68%/queries |
|                                             | Index Nested-Loop Join                        | C                 | 42.11%/queries |
|                                             | Tag-Array Join                                | E                 | 26.32%/queries |
|                                             | Hash Join                                     | C                 | 10.53%/queries |
|                                             | Sort-Merge Join                               | C                 | 2.63%/queries  |
| Aggregation Strategies                      | Inline (Fused) Aggregation                    | E                 | 97.37%/queries |
|                                             | Dense-Key Aggregation                         | E                 | 55.26%/queries |
|                                             | Scalar Aggregation                            | C                 | 39.47%/queries |
|                                             | Two-Phase Aggregation                         | C                 | 39.47%/queries |
|                                             | Direct Array Aggregation                      | E                 | 28.95%/queries |
|                                             | Hash Aggregation                              | C                 | 18.42%/queries |
| Low-Level Execution Optimizations           | Pointer <code>__restrict</code> Aliasing      | E                 | 50.00%/queries |
|                                             | Integer Scaled Arithmetic                     | E                 | 44.74%/queries |
|                                             | Branchless Range Check                        | E                 | 18.42%/queries |
|                                             | Branch-Hint ( <code>__builtin_expect</code> ) | E                 | 13.16%/queries |
|                                             | Manual Loop Unrolling                         | E                 | 13.16%/queries |
|                                             | Software Prefetching                          | E                 | 10.53%/queries |

**Table 2: Overview of bespoke strategies employed in the storage layout and query-processing implementations across Bespoke-TPCH and Bespoke-CEB. We classify the strategies into classical (C) and exotic (E). Classical ones can also be found in general-purpose OLAP engines, but here they are used in a workload-specific manner. Exotic ones are strategies that are not used in relational engines.**

operators, aggregation strategies, and low-level execution optimizations. At the storage level, the agent applies dictionary encoding to a quarter of all columns, physical sort orders to nearly 19%, and compact representations such as scaled integers and 16-bit date offsets where applicable. On the query-execution side, bitmap semi-joins (73.7% of queries) and inline fused aggregation (97.4%) emerge as the most prevalent strategies, while hash indices for  $O(1)$  join lookups and precomputed derived columns appear in 47.4% and 36.8% of queries, respectively. At the lowest level, pointer aliasing hints (`__restrict`) and integer-scaled arithmetic are applied in roughly half of all queries, complemented by branchless range checks, branch hints, and software prefetching where beneficial. To illustrate how these bespoke and non-bespoke strategies manifest in concrete queries, we highlight two representative examples:

**TPC-H Query 5.** Q5 exploits the physical sort order of the *orders* table by *orderdate* to bound scans via *lower\_bound*, and uses a per-order range directory to touch only the relevant *lineitem* slices, a strategy that benefits further from *lineitems* being sorted by *orderkey*. Denormalized attributes, i.e., *lineitem.supp\_nationkey*, and *discounted\_price*, are stored directly next to *orders.cust\_nationkey*, rather than in separate tables, eliminates extra joins and redundant computations in the inner loops. Region and nation lookups reduce

to simple  $O(1)$  array indexing via *region.name\_to\_key* and dense key-to-nation arrays.

**TPC-H Query 13.** Q13 performs a focused two-column scan, touching only *orders.custkey* and *orders.comment* (plus *customer.custkey* during aggregation). Variable-length comments are accessed through a string arena (offsets + data) to enable fast offset-based substring checks. Crucially, each order row carries two bitmasks: *alpha\_mask*, recording which letter appear in comment, and *bigram\_mask*, recording hashed character pairs, that cheaply rule out non-matching rows before any substring search is attempted.

**Discussion of Results.** While the Bespoke Agent draws from a large vocabulary of techniques, its selections are remarkably *targeted*: for each query, it typically applies only a small, carefully chosen subset of strategies. This query-specific, and even column-specific, decision-making is precisely what distinguishes a bespoke engine from a general-purpose system, which must rely on broadly applicable but less optimal strategy choices. We provide the generated code alongside our Bespoke-OLAP framework.<sup>4</sup>

## 5.4 The Synthesis Process

We now shift our perspective from the *output* of the Bespoke-OLAP agent to the *process* by which the Bespoke Agent arrives at its implementations. Understanding the synthesis, including the development trajectory, tool usage patterns, and correctness behavior, is essential for assessing the practical viability of the approach.

**Development Trajectory and Speedups over Time.** Figure 9 traces the synthesis process of Bespoke-TPCH from start to finish. Over the course of approximately 15,000 agent turns, the agent progressed through all stages, starting from 5,700 lines of code and an initial speedup of 2.34 $\times$  over DuckDB, and growing to a final engine of approximately 11,600 lines of C++, of which more than 3,000 lines are tracing infrastructure (hidden behind `#ifdef` directives). Occasional regressions (e.g. turn 11,600) during optimization were quickly detected and resolved via rollbacks. Context demands grew substantially in later stages, peaking at over 150,000 tokens, with regular compaction preventing context limit violations.

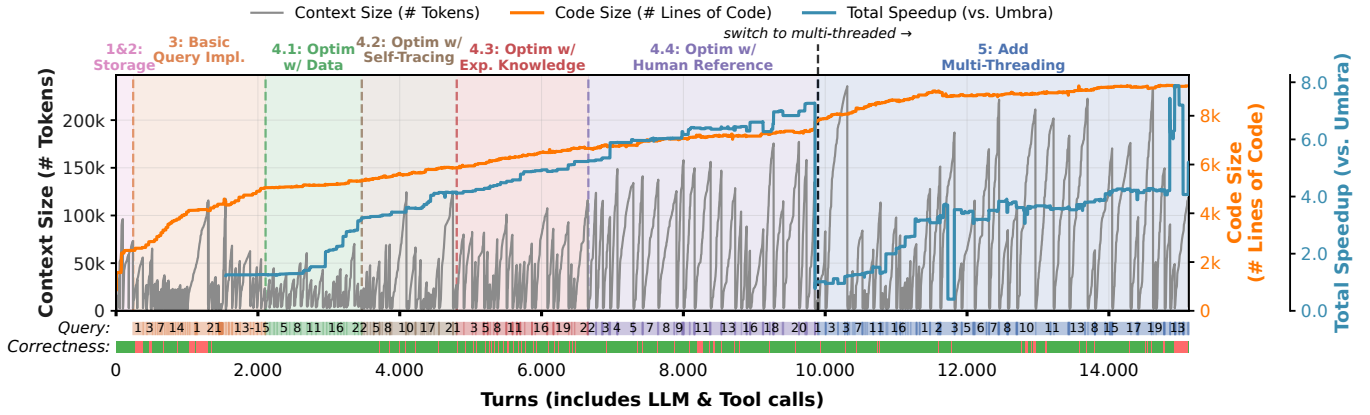
**Synthesis Cost.** The total cost of synthesizing Bespoke-TPCH amounted to \$276.44 in API costs (including \$251.02 for the optimization stages), while Bespoke-CEB required \$303.25 (\$259.26 for optimization).<sup>5</sup> The required wall-clock time ranges from 10-30 hours (including MT), depending on the LLM usage tier, rate limits, and queuing time. Given that the resulting engines deliver order-of-magnitude speedups over DuckDB on their respective workloads, these one-time synthesis costs compare favorably to the cumulative savings in query execution time, particularly for workloads that are executed repeatedly over extended periods.

## 6 RELATED WORK

**Manual Specialization.** A large body of research shows that specialization yields substantial performance gains. Vectorized and data-centric processing, as in MonetDB / X100 [5], mitigated the costs of generality by adapting query execution to data-intensive OLAP workloads, and column stores and main-memory systems such as HyPer [12] further demonstrate that restricting generality improves performance. However, these systems specialize at the

<sup>4</sup>Generated code: [https://github.com/DataManagementLab/BespokeOLAP\\_Artifacts](https://github.com/DataManagementLab/BespokeOLAP_Artifacts)

<sup>5</sup>These costs reflect the Claude Sonnet 4.7 pricing at the time of writing.



**Figure 9: Development timeline of Bespoke-TPCH.** The upper plot shows context size (left axis) and cumulative lines of code (right axis) over agent turns. Below the chart, the query currently being worked on is indicated, with green and red markers denoting correctness status. Compaction sessions periodically reduce the context size to prevent exceeding context limits.

workload-class level rather than the individual-workload level; constructing a fully bespoke engine for a specific deployment remained a manual engineering effort until this paper.

**Query Compilation.** Query compilation further advanced specialization by generating executable code tailored to individual queries, eliminating interpretation overhead and enabling aggressive inlining and loop fusion. HyPer [12, 20], LegoBase [14], and DBToaster [4] demonstrated the effectiveness of compiled query pipelines, and modern engines such as DuckDB [23] and Umbra [21] adopt similar principles. Despite these advances, query compilation remains restricted to the execution layer and targets general SQL rather than a small, fixed query set.

**DBMS Tuning.** A dominant form of deployment-specific optimization is *physical database design*, which selects workload-driven data structures such as indexes, clustering, partitioning, and materialized views. Extensive work has automated these decisions, from early cost-driven index selection and *what-if* analysis [6, 7] to integrated views, partitioning, and commercial tuning advisors [1–3]. A complementary line of work focuses on configuration tuning and self-driving databases [11, 22, 28], as well as learned models for indexes, cardinality estimation, and cost models [13, 15, 18].

These methods can improve performance but are confined to a fixed architecture, whereas synthesizing a workload-specific DBMS enables deeper, architecture-level optimizations.

**LLM-Driven Synthesis.** Large language models enable automated generation of nontrivial software artifacts [8], and tool-augmented agents can iteratively refine code using execution feedback [33]. CodexDB [26] generates query-processing code from SQL queries and natural-language instructions, and GPT-DB [27] generates customizable SQL processing code. However, neither aims for a bespoke design. Program synthesis systems such as FFTW [9] and ATLAS [32] similarly generate high-performance implementations automatically. Existing approaches target individual programs, kernels, or query-specific logic, however, and do not construct coherent systems with tightly coupled storage, execution, and architectural components, whereas database engines exhibit strong cross-layer dependencies requiring structured construction and optimization across interacting subsystems.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we presented *Bespoke OLAP*, a fully autonomous synthesis pipeline that leverages LLM-guided code generation to produce complete, workload-specific database engines from scratch. On TPC-H (SF20), it achieves **11.17×** over DuckDB v1.4.1 and **7.24×** over Umbra; on CEB (SF2), it achieves **45.33×** and **9.56×**, respectively. These results indicate that a substantial fraction of the overhead in modern analytical systems arises from their requirement to remain general-purpose, and that LLM-guided synthesis can systematically eliminate this cost on a per-workload basis. Because each engine is synthesized within hours at a cost of only a few dollars, per-workload specialization, which historically was reserved for exceptional, high-value deployments, can plausibly become routine for future DBMS-systems.

For future directions, there are many interesting challenges and opportunities. While our current submission already supports single-threaded disk-based execution, the achieved speedups do not yet reach the level of the in-memory setting. Closing this gap will require synthesizing bespoke buffer pool management, eviction policies, and block-aligned page layouts tailored to the known query access pattern. We consider this an important direction, though one that introduces a substantially larger synthesis problem which we will tackle in the future. Beyond disk-resident workloads, incorporating lightweight cost models tailored to LLM-generated code could further reduce synthesis time by pruning unpromising optimization alternatives early, particularly for join ordering. Finally, extending bespoke synthesis beyond pure OLAP to other workloads such as OLTP introduces consistency and isolation guarantees as a fundamentally new dimension of the synthesis problem.

## ACKNOWLEDGMENTS

This research was partially funded by the state of Hesse as part of the NHR program (NHR4CES), the LOEWE Spitzenprofessor of the state of Hesse (III 5-519/05.00.003-(0005)), the project "hessian.AI AISC" (Grant No. 01IS22091) by the Federal Ministry of Education and Research (BMBF), as well as by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC3057/1 "Reasonable Artificial Intelligence", Project No. 533677015). We also thank DFKI Darmstadt and hessian.AI for their support.

## REFERENCES

- [1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.
- [2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.
- [3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 359–370.
- [4] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment* 5, 10 (2012), 968–979.
- [5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*. 225–237.
- [6] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin “what-if” index analysis utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.
- [7] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. San Francisco, 146–155.
- [8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [9] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP’98 (Cat. No. 98CH36181)*, Vol. 3. Ieee, 1381–1384.
- [10] Joran Dirk Greef. 2024. Rediscovering Transaction Processing from History and First Principles. Retrieved February 21, 2026 from <https://tigerbeetle.com/blog/2024-07-23-rediscovering-transaction-processing-from-history-and-first-principles/>
- [11] Benjamin Hilprecht, Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thosttrup, and Tobias Ziegler. 2020. DBMS Fitting: Why should we learn what we already know?. In *CIDR*. [www.cidrdb.org](http://www.cidrdb.org).
- [12] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.
- [13] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).
- [14] Yannis Klonatos, Christoph Koch, Tiark Rumpf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.
- [15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
- [16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
- [17] Ryan Marcus. 2023. Learned Query Superoptimization. In *Proceedings of the AIDB 2023 Workshop*.
- [18] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).
- [19] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2019–2032.
- [20] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550. <https://doi.org/10.14778/2002938.2002940>
- [21] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12–15, 2020, Online Proceedings*. [www.cidrdb.org](http://www.cidrdb.org). <https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html>
- [22] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.
- [23] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.
- [24] Michael Stonebraker and Ugur Çetintemel. 2005. “One Size Fits All”: An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2–11. <https://doi.org/10.1109/ICDE.2005.1>
- [25] Transaction Processing Performance Council. 2022. TPC Benchmark H (Decision Support) Standard Specification. Retrieved July 20, 2026 from [https://www.tpc.org/TPC\\_Documents\\_Current\\_Versions/pdf/TPC-H\\_v3.0.1.pdf](https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf) Version 3.0.1.
- [26] Immanuel Trummer. 2022. CodexDB: Synthesizing code for query processing from natural language instructions using GPT-3 Codex. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2921–2928.
- [27] Immanuel Trummer. 2023. Demonstrating gpt-db: Generating query-specific and customizable code for sql processing with gpt-4. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4098–4101.
- [28] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.
- [29] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. (2024).
- [30] Johannes Wehrstein, Tiemo Bang, Roman Heinrich, and Carsten Binnig. 2025. GRACEFUL: A Learned Cost Estimator for UDFs. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19–23, 2025*. IEEE, 2450–2463. <https://doi.org/10.1109/ICDE65448.2025.00185>
- [31] Johannes Wehrstein, Roman Heinrich, Mihail Stoian, Skander Krid, Martin Stemmer, Andreas Kipf, Carsten Binnig, and Muhammad El-Hindi. 2025. Redbench: Workload Synthesis From Cloud Traces. *arXiv preprint arXiv:2511.13059* (2025).
- [32] R Clint Whaley, Antoine Petit, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel computing* 27, 1–2 (2001), 3–35.
- [33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. [n.d.]. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.