Cloud Analytics Benchmarking: Where We Are, What Is Missing, and Where We Should Go

Michail Georgoulakis Misegiannis*
Technische Universität München
geom@in.tum.de

Till Steinert*
Technische Universität München
till.steinert@tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

Muhammad El-Hindi
Technische Universität München
muhammad.el-hindi@tum.de

Abstract

Workload traces from cloud vendors have significantly advanced our understanding of real-world analytical workloads. These traces contain execution telemetry (CPU time, scanned bytes, operator counts) but omit SQL text for privacy reasons, preventing their direct use as benchmarks. This raises a fundamental question: *Can realistic workloads be synthesized from anonymized telemetry?* We argue that current telemetry-based synthesis can capture workload-level properties well (e.g., concurrency, repetition) but remains challenging at the query level. We show that telemetry varies across systems, deployments, and execution contexts, and does not uniquely identify the query structure. To bridge this gap, we outline two complementary paths: enriching traces with engine-agnostic *transferable workload features*, and building *use-case-centric benchmarks* directly from open application scenarios such as ELT pipelines and BI dashboards.

1 Introduction

Traditional benchmarks do not capture modern workloads. Database benchmarking has long relied on hand-crafted workloads such as TPC-H, TPC-DS, and more recent industry and academic benchmarks, including ClickBench [24] and JOB [15]. These benchmarks provide consistent and reproducible testbeds for system comparison, but they cover only a limited set of query patterns shaped by the benchmark designer's assumptions and domain knowledge. Meanwhile, evolving warehouse trends (e.g., ETL versus ELT), new data sources such as data lakes [7, 8], and emerging workload classes such as agentic AI increasingly challenge the assumptions embedded in traditional benchmarks.

Industry traces offer insights into real workloads. Anonymized workload traces released by Snowflake [33], and AWS [31] have significantly advanced our understanding of real-world analytical workloads, revealing patterns such as skewed resource consumption, long-tail behavior, query repetition, and string-heavy schemas that traditional benchmarks fail to capture. To preserve privacy, these traces expose execution telemetry (e.g., arrival times, CPU time, bytes scanned) rather than raw SQL text, but even this partial view has enabled useful workload characterization and motivated a new generation of workload synthesis tools.

Trace-based query synthesis. A growing body of work builds on these traces by synthesizing executable queries whose telemetry matches a target workload profile. As we review in Section 2, these approaches use techniques from Bayesian optimization and template pools to LLM-based generation loops [12–14, 32, 34–36, 39]. Their promise is compelling: rather than relying on a handful of hand-crafted benchmarks, one could generate diverse workloads grounded in production behavior as captured in the traces.

Limits of telemetry-based synthesis. However, as we show in this paper, while telemetry has enabled the creation of workloads that mimic workload-level trends, currently synthesized queries may diverge structurally from the original queries. Our observation is that current traces capture little of the application context behind queries, leaving telemetry-based synthesis largely a *black-box* process that cannot directly reason about whether generated queries faithfully represent workload queries beyond their telemetry. Enriching traces with more detailed query characteristics could narrow this gap, but it is currently constrained by what industry can safely share without exposing sensitive customer information.

Vision: Closing the workload gap. To address this, we envision two complementary paths. First, we call on industry to enrich published traces with richer, engine-agnostic workload features beyond raw telemetry (Section 4.1). Second, for the research community, we propose *use-case-centric benchmarks* built directly from open application scenarios such as ETL pipelines and Business Intelligence (BI), where queries are transparent by construction rather than reverse-engineered. As a proof of concept (Section 4.2), we show that open-source tools such as dbt (ELT) and Apache Superset (BI) can generate workloads with characteristics found in real traces, such as string- and join-heavy queries, while exposing the actual SQL rather than only telemetry. We illustrate this with an e-commerce revenue analysis based on the dbt Jaffle Shop project [3].

2 Background and Related Work

With continuously changing workloads and requirements, there is a rich body of work on advancing database benchmarking [4, 5]. Related work also considers database testing [21, 37]. In the following, we focus on workload traces and synthesis approaches most relevant to benchmark design. Other workload traces [17], trace-analysis tools [6], and query generation methods [23, 26] are related, but orthogonal to our focus.

*These authors contributed equally to this work

This work is licensed under a Creative Commons Attribution 4.0 International License.
DBTest '26, Bengaluru, India
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2701-6/2026/05
https://doi.org/10.1145/3810991.3811634

2.1 Telemetry-based Workload Traces

To date, two major cloud database providers have published telemetry workload traces that contain operational measurements of system state during query execution. Although these traces reveal similar high-level patterns, they differ in what they expose.

Snowset. Snowset contains query-level statistics for approximately 69 million queries executed on the Snowflake cloud data warehouse over two weeks in early 2018 [33]. It reports both query-level and operator-level execution statistics, including runtimes and resource usage. Its analysis shows that the operator mix is more balanced than in traditional analytical benchmarks: only about 50% of execution time is spent in scans and filters, compared with about 84% in single-node TPC-H. It also reveals highly skewed resource consumption, with a small fraction of queries accounting for most CPU time, as well as a large number of read/write (ETL-style) jobs.

Redset. Redset spans three months of query telemetry collected from AWS Redshift instances [31]. Compared with Snowset, it provides fewer execution statistics and less operator-level detail, but exposes richer metadata, including structural features such as join and aggregation counts. It reveals a similar skew: fewer than 0.1% of queries account for roughly 25% of all resource usage.

These traces confirm that production workloads differ fundamentally from traditional benchmarks in terms of diversity, skew, and operator mix.

2.2 Telemetry-based Query Synthesis

As discussed in the introduction, a growing body of work derives executable benchmarks by synthesizing queries whose telemetry matches workload traces, aiming to produce more realistic workloads than hand-crafted benchmarks. We broadly classify these approaches into matching-based and LLM-based methods.

Matching-based approaches. Matching-based approaches draw from a pool of existing queries, whether from standard benchmarks such as TPC-H and TPC-DS, user-provided workloads, or parameterized templates, and select candidates whose profiles best align with the target telemetry. Stitcher [34] uses a prediction model to select candidate queries for a target performance profile, independent of Snowset or Redset. The Cloud Analytics Benchmark [32] focuses on generating query streams that match higher-level workload properties in Snowset, such as tenant mix and burstiness. PBench performs more fine-grained synthesis by partitioning workloads into time windows and composing queries to match aggregated execution statistics [39]. Redbench [12, 36] goes a step further with per-query synthesis from both query pools and templates, preserving query-level patterns such as temporal order, repetitions, scan-set similarity, and mixed read/write behavior. These systems make workload reconstruction practical. Their primary success metric is telemetry alignment, leaving open how closely the generated queries resemble the originals structurally.

LLM-based approaches. Recent systems increasingly use LLMs to increase the structural fidelity of the generated queries. For example, PBench includes a mode that uses an LLM to enrich queries from its query pool. SQLBarber addresses the diversity problem by using an LLM to generate SQL templates and a feedback-guided search procedure to instantiate queries that match target cost distributions [13, 14]. ResQ [35] uses LLMs to translate structure-aware internal representations into executable SQL, and combines this step with local performance models and query reuse to better match per-query targets such as CPU time and scanned bytes.

Telemetry alignment, but query-level mismatch. The above approaches show that telemetry alignment is a useful objective for generating benchmarks that better reflect production workload characteristics. In particular, the synthesized workloads can reproduce aggregate properties such as CPU utilization, bytes scanned, and join counts. However, we observe clear mismatches at the query level, including differences in query structure and the operators exercised. As we show next, through a per-query analysis of the synthesized workloads, a query or query set may satisfy the telemetry objective, even if it does not resemble the underlying ground-truth query in terms of intent, structure, or execution behavior.

3 Challenges of Telemetry-based Synthesis

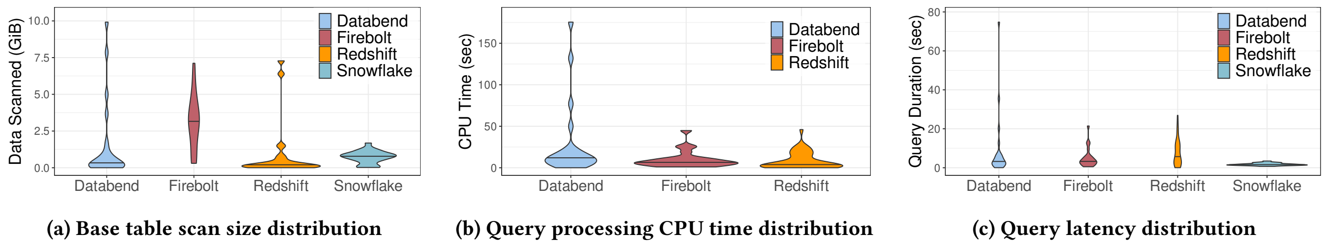
Telemetry alignment is a useful and achievable objective at the workload level, but it is not sufficient for faithfully reconstructing modern workloads. In many cases, capturing workload behavior requires preserving the properties of individual queries. For example, industry reports [30, 31] show long tails of operator counts and nesting structures that trigger different code paths and optimization decisions within a database system. A synthesis method that optimizes only for telemetry may therefore match observed metrics while still producing different execution behavior. This limitation arises because workload telemetry depends on several factors beyond query text alone. We illustrate this problem through three examples: the execution system, the execution context, and the ambiguity in the mapping from query structure to telemetry.

3.1 System-dependent Challenges

Telemetry-guided synthesis assumes that the telemetry observed during generation is comparable to the telemetry recorded in the target trace. In practice, however, this is rarely possible: Redset traces originate from managed AWS Redshift clusters and Snowset from Snowflake's managed service, neither of which can be fully replicated by external researchers who must rely on different engines or locally deployed systems. This raises the question: *how much do the choice of engine and hardware affect the telemetry?*

1. Same hardware, different engine. To analyze this, we execute TPC-H at scale factor 20 on four analytical systems: Databend, Firebolt, Redshift, and Snowflake, using identical schema, data, and query templates, adapted only where each engine's SQL dialect requires it (e.g., date arithmetic). The four engines cover two deployment models and distinct execution strategies: Databend (open-source) and Firebolt [20] are self-hosted, while Redshift and Snowflake are managed services. Redshift compiles queries to native code; the others use vectorized execution. To isolate the engine's effect, we align hardware as closely as possible: Redshift on a single-node *ra3.xlplus*, Snowflake on a Gen2 XSMALL warehouse, and the remaining systems on an *i4i.xlarge* instance (4 vCPUs, 32 GiB RAM) matching *ra3.xlplus*.

Different engines produce different telemetry. Figures 1a to 1c show that the resulting telemetry profiles differ substantially across systems, even though the logical workload is identical. Base-table scan size (Fig. 1a), for example, depends on physical data layout



(a) Base table scan size distribution **(b) Query processing CPU time distribution** **(c) Query latency distribution**

Figure 1: Telemetry distributions for the same TPC-H SF20 workload differ substantially across four analytical engines. Because telemetry is system-dependent, matching it on one engine does not guarantee recovery of the original workload on another.

choices such as partitioning, compression, and indexing, as well as whether the engine reports logical or physical bytes read. Similar issues arise for CPU time and wall-clock latency. CPU time (Fig. 1b) depends on execution engine efficiency, code generation, vectorization, parallel scheduling, and I/O stalls. Latency (Fig. 1c) also accounts for synchronization and queueing overheads. As a result, the same query can appear CPU-heavy on one engine and I/O-bound on another. Telemetry, therefore, describes how a particular engine executed a query, not the query itself.

2. Same engine, different hardware. The previous experiment isolated the engine effect by aligning the hardware. In practice, this cannot always be guaranteed. Managed cloud services run on opaque and often heterogeneous infrastructure: Snowflake warehouses may draw from multiple instance types [28], while Redshift clusters run on specialized *ra3* nodes that are not available as standard EC2 instances. This hardware information cannot be reconstructed from the dataset, although processor generation, hardware architecture (e.g., x86 versus ARM), cluster size, parallelism, and data locality all influence the amount of scanned bytes, CPU utilization, and runtime [27]. Thus, even within the same DBMS, telemetry collected on one deployment may not transfer reliably to another.

Different hardware produces different telemetry. Figure 2 illustrates this effect for Firebolt on three EC2 instance types: *i4i* (x86, Ice Lake), *i7i* (Intel Sapphire Rapids, x86), and *i8g* (AWS Graviton4, ARM). All three provide the same number of vCPUs and memory, yet the runtime differs substantially. Moving from *i4i* to *i7i*—a newer x86 generation, both with hyperthreading—yields a modest ~20% improvement in mean latency. In contrast, *i8g* roughly halves both median and maximum latency compared to *i4i*, suggesting that much of the gap stems from the use of physical cores (no hyperthreading) rather than CPU generation alone.

> **Insight 1** Telemetry is specific to the engine and hardware that produced it; the same workload can yield different telemetry across systems and configurations.

3.2 Execution Context Challenges

Beyond engine and hardware differences, telemetry also depends on the conditions under which a query executes. In production, queries share CPU, memory, and I/O bandwidth with other concurrent queries from the same or different tenants [18]. This concurrency can be substantial: some warehouses in Snowset contain periods with more than 100 active queries [33]. Additionally, cache state from prior executions can accelerate or penalize subsequent queries. While some of these effects are observable in traces (e.g., via timestamps or queue depth), the telemetry metrics used as synthesis targets (CPU time, scanned bytes, runtime) blend the query's inherent cost with the context in which it ran. As a result, the synthesis target for the same query depends on when it runs, making it an unstable optimization objective. This raises a second question: *how much does execution context affect the telemetry of a query?*

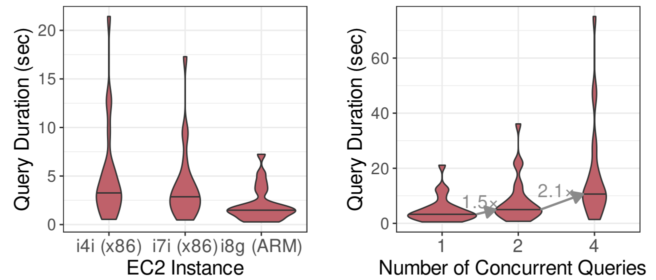


Figure 2: TPC-H runtime on Firebolt varies across EC2 instance types (left) and concurrency on *i4i* (right), even for identical queries. All instances use 4 vCPUs and 32 GiB RAM.

Contention changes telemetry. To quantify this effect, we execute TPC-H queries at scale factor 20 on Firebolt running on an *i4i.xlarge* instance. For each experiment, we duplicate the same query N times and submit all copies concurrently, thereby varying contention while keeping query semantics fixed. Figure 2 shows that latency increases substantially with concurrency. Even median runtime shifts noticeably: increasing concurrency from one to two copies raises median latency from 3.3 s to 4.99 s, and increasing to four copies raises it further to 10.68 s. The tail is even less stable: under four-way concurrency, maximum latency is more than 3× higher than in isolation.

Cache state changes telemetry. Caching introduces another source of instability. Repeated executions may benefit from warmed result, data, and metadata caches, or from reused intermediate state, while concurrent writes may invalidate caches and degrade subsequent reads. Figure 3 quantifies this effect: The cold versus warm execution of TPC-H on the previous single-node Redshift cluster at scale factor 10 shows that warm runs reduce median latency by roughly 2× and up to 4× for individual queries. As a result, two executions of the same query under identical logical conditions can produce different telemetry depending on recent workload history.

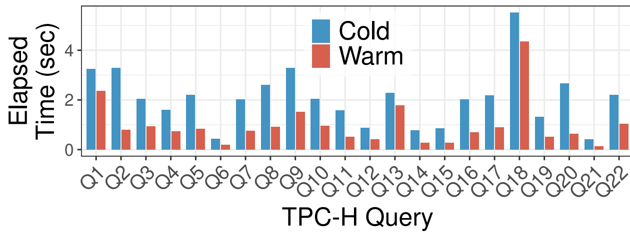


Figure 3: Cold and warm executions of the same TPC-H queries (SF 10) exhibit substantially different runtimes, demonstrating the impact of cache state on telemetry.

The importance of caching effects is further highlighted by recent work on result caching [22] and benchmarking [12]. For example, Redbench [12] attempts to capture such effects by preserving query repetition and temporal order, or by leveraging Redset's explicit `was_cached` information. However, Redbench's approach requires query fingerprints or explicit caching annotations in the trace, information that, for example, Snowset does not provide.

> **Insight 2** Telemetry reflects not just the query but its surrounding execution context; faithful synthesis should account for these effects.

3.3 Structure Reconstruction Challenges

Even after accounting for the above factors (deployment, caching, and isolation), we observe a fundamental challenge: Multiple structurally different queries can produce similar execution telemetry.

Query-to-telemetry mapping is not injective. Figure 4 illustrates this phenomenon using two TPC-H-like queries. Query A joins `orders` and `lineitem`, whereas Query B adds a three-table join chain and an additional grouping dimension. Despite these structural differences, both queries produce nearly identical telemetry. That is because Query A derives selectivity from its filter on `lineitem`, while Query B derives selectivity from its join conditions. More generally, query-to-telemetry mappings are *many-to-one*: a synthesis method may match a telemetry target perfectly while still generating a query that is structurally unrelated to the original. This motivates the question: *to what extent can telemetry-based synthesis capture the underlying queries of a workload?*

Experiment: matching-based synthesis. To answer this question, we conduct a controlled experiment using TPC-H as the ground-truth workload, allowing us to test whether telemetry-matched queries are also structurally similar. This is a deliberately conservative test: if current synthesis methods diverge from these known and well-studied queries, they are unlikely to recover more complex queries documented in traces. We study two representative synthesis approaches: one matching-based and one LLM-based. We first evaluate the matching-based synthesizer PBench [39] by providing it with a telemetry trace derived from our TPC-H workload. In its matching-based mode, without LLM-assisted generation, PBench selects queries from a fixed pool, including the TPC-H queries, to match the telemetry of target time windows. We then compare the synthesized workloads with the original workload under both isolated and concurrent execution.

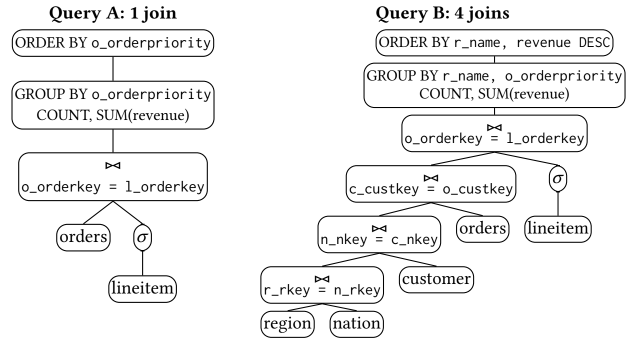


Figure 4: Two structurally different queries with nearly identical telemetry. Query A joins only `orders` and `lineitem`, whereas Query B adds a `region–nation–customer` join chain and an additional grouping dimension.

Aggregate statistics are similar, queries are not. At the distribution level, PBench closely matches the target: The generated time windows exhibit CPU time and scanned-byte volumes similar to those of the original workloads, confirming that telemetry alignment is achievable and meaningful. However, at the query level, the picture is more nuanced. Even in a controlled setting (sequential TPC-H at SF 20 with no concurrency), 4 of the 22 benchmark queries are matched by structurally different queries from PBench's pool. For instance, Q8 (an 8-table join with `CASE` and a subquery) is matched to Q13 (a 2-table outer join with `LIKE`). Although these queries produce similar telemetry, they exercise different operators and code paths. This result shows that telemetry alignment, while effective at the workload level, does not guarantee structural fidelity at the query level because the query-to-telemetry mapping is many-to-one. Furthermore, matching-based approaches are constrained by their query pool: if the pool lacks a structurally similar query, telemetry matching alone cannot recover the original structure.

Experiment: LLM-based synthesis. In principle, LLM-based generators can narrow the gap between the target workload and the matching based approaches by producing arbitrary candidate queries rather than selecting from a fixed pool. They can also be guided by structural signals beyond telemetry, such as aggregation patterns or predicate structure, thereby reducing the candidate space. This flexibility allows researchers to inject domain knowledge directly into the synthesis process.

ResQ to the rescue? To evaluate whether LLM-based query synthesis achieves better structural fidelity, we analyze the output of ResQ [35], the most feature-aware synthesis approach to date. ResQ conditions generation on structural targets, such as table selection, join count, aggregation count, and sort order. It uses a multi-stage pipeline: an LLM generates a SQL template matching these targets, and a Bayesian optimization loop tunes predicate thresholds until CPU time falls within 20% of the target. We run ResQ on TPC-H and generate three candidates per query (66 in total), of which 58 execute successfully. To assess structural fidelity, we compare the 58 synthesized queries with the 22 original TPC-H queries using 27 Abstract Syntax Tree (AST)-based structural features (see Appendix A). We measure similarity using standardized Euclidean distance so that features on different scales contribute comparably.

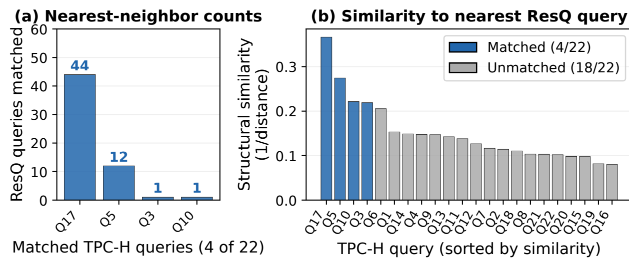


Figure 5: Structural coverage of TPC-H by 58 LLM-generated queries (27 features, standardized Euclidean distance). (a) The generated queries cluster around only 4 of the 22 TPC-H queries as nearest neighbors. (b) Even the best matches remain structurally dissimilar, and 18 of 22 TPC-H queries are not the nearest neighbor of any generated query.

LLM-synthesized queries lack structural diversity. Figure 5 shows that the generated queries cluster as nearest neighbors to just 4 of the 22 TPC-H queries, suggesting that the current feature set guides generation toward a narrow subset of the structural space. ResQ matches its targeted features well: join counts, aggregations, and sort order align with the respective TPC-H queries while simultaneously achieving similar telemetry metrics. However, structural features outside its scope of optimization are entirely absent from the 58 generated queries. In particular, subqueries, Common Table Expressions (CTEs), `CASE`, `HAVING`, `BETWEEN`, and `LIKE` are present in TPC-H but absent from the synthesized SQL. AST depth in the generated queries is also lower (mean 5.4 vs. 11.4), and `GROUP BY` clauses are broader than typical (mean 14.1 columns vs. 1.5) to include injected, artificial compute expressions that help meet CPU-time targets (see Appendix B). This reveals a broader pattern: ResQ reproduces the features it explicitly targets (# of joins, aggregations), but diverges on those it does not model. This suggests that telemetry alignment alone is unlikely to reconstruct broader query-level structure unless the synthesis objective explicitly models it.

> **Insight 3** Telemetry alignment alone is insufficient to reproduce query-level characteristics in workloads. Expanding the set of structural features used for synthesis can help narrow the space of candidate queries.

4 Two Paths to Realistic Benchmarks

Realistic workloads are mutually beneficial. Realistic benchmark workloads are a shared interest across the database community. Academic research needs them to evaluate systems on problems that matter in practice; industry benefits when academic techniques apply to production workloads. The challenges above show that telemetry, as currently exposed, cannot alone bridge this gap. At the same time, the industry cannot simply open-source the queries themselves for privacy reasons.

We envision two complementary paths forward, each a work in progress, with early thoughts and findings below. One (Section 4.1) calls on industry to expose richer, transferable workload features beyond raw telemetry. The other (Section 4.2) builds workloads directly from open use cases, independent of production traces.

4.1 Toward Transferable Workload Features

The true bottleneck: A conflict of interest. As shown in Section 3.3, the bottleneck of workload generation is the availability of high-quality structural query features lacking in current traces. Although cloud deployments capture much more detailed telemetry [37], industry has a legitimate interest in restricting workload insights as publishing raw SQL text or query plans could leak personally identifiable information and reveal internal optimizations.
The quest for a new trace format. The challenge is to find a trace format that abstracts from concrete system implementations but is sufficiently rich to allow the reconstruction of structurally faithful queries. In particular, we advocate incorporating *transferable workload features*, i.e., features that preserve workload-relevant behavior across engines and deployment settings without exposing raw queries. Inspired by transferable database features [9], we identify four categories of transferable workload features that can capture query-level behavior in an engine-agnostic manner:

- **Query structure:** join count and shape, aggregation count, subquery depth, CTE usage, window functions. These can be derived from any SQL AST and are fully engine-agnostic.
- **Data characteristics:** tuple width, column cardinality, null fraction, data-type distribution. These can be obtained from any system catalog without revealing customer data.
- **Plan-shape features:** join algorithm family (hash, merge, nested-loop), parallelism degree, and relative cardinalities. Although partially system-dependent, we can further abstract to engine-neutral descriptions, such as "equi-join with large build side".
- **Workload-level features:** query arrivals, concurrency level, read/write ratio, table access frequency, and session structure (e.g., ELT pipeline DAG depth versus ad hoc exploration).

Recent attempts of standardizing system-agnostic query representations (e.g., Substrait [1]) could lay the foundation of this new trace format. However, query anonymization and encoding of data characteristics (e.g., replacing filter predicates while preserving selectivity) remain open challenges.

4.2 Vision of Use-Case-Centric Query Synthesis

When the use case one wants to model or optimize against is known, realistic workloads can be built directly from it rather than reverse-engineered from partial execution observations. A *use-case-centric* workload begins with a well-defined analysis task, identifies the relevant questions, and lets standard data-engineering tools (e.g., dbt, Superset) express them as SQL queries. We illustrate this with a proof-of-concept based on dbt Jaffle Shop, a synthetic e-commerce dataset. The prototype spans the full data-engineering lifecycle (Figure 6): ingestion, interactive exploration, and serving.
Ingesting data into the warehouse. Before data scientists can explore and analyze the data, it must first be ingested into the data warehouse (Figure 6 ①). We assume that the data sources already exist outside the system (e.g., in object storage or an ERP system) and are imported through an ELT pipeline. This is an important departure from previous benchmarks like TPC-DS, which assume data is already normalized and stored in the database. However, we argue that data pipelines are an important component of real-world analytics, accounting for 54.7% of Redset query runtime [31]. In our example, we use dbt to ingest the shop data into Firebolt.

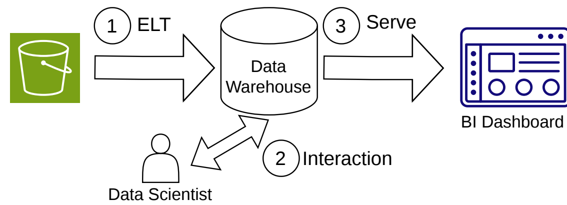


Figure 6: Blueprint of a use-case-centric workload spanning ELT ingestion, interactive analysis, and serving through dashboards and downstream applications.

Interactive data exploration. Next, data scientists will familiarize themselves with the dataset through exploratory, ad hoc queries ② (e.g., calculating aggregate statistics). They then formulate concrete questions to understand customers' purchasing behavior. In the Jaffle Shop example, these questions might include the following:

- **Q1:** Who are the top 10 customers with the highest customer lifetime value? How much revenue do they account for?
- **Q2:** Are there seasonal trends in revenue generation?
- **Q3:** What products are often bought together?

These questions are then answered through iterative query refinement. For example, Q3 can be answered by self-joining `order_items` on `order_id` and counting co-occurrences of product pairs.
Serving and monitoring. Finally, data engineering projects are often not a one-off analysis, but must be served and maintained ③. This effort usually includes continuous ingestion of new data via orchestration tools such as Apache Airflow, periodic reporting queries, KPI dashboards, and data-driven applications such as product recommendation systems. In our example, the data scientists create interactive KPI dashboards for Q1–Q3 using the open-source business intelligence tool Apache Superset.
Preliminary workload insights. To analyze the prototype, we exported all SQL statements executed during the use case from Firebolt's `engine_user_history` view, keeping only generated queries by dbt and Superset to avoid SQL-skill bias. Our analysis suggests that the use-case-centric workload shares similarities with industry telemetry traces and exhibits patterns absent from existing benchmarks. Compared with ResQ, the queries are structurally more complex and include features such as CTEs, `LIKE`, and `BETWEEN`. We also observe that dbt compiles its internal DAG into a sequence of `CREATE TABLE AS SELECT` (CTAS) statements, consistent with the hypothesis of van Renen et al. [31] that CTAS is common in data pipelines. The query patterns collected from our limited sample align with observations from industry workload studies [30]: Our trace exhibits joins on string columns, use of timestamp operations, window functions, and `CASE WHEN` expressions. Structural repetition is common, with the same query template often instantiated with different filter values. Finally, `GROUP BY` columns are predominantly text, date, or timestamp attributes, consistent with the string-heavy profile reported for Snowflake BI workloads.
Fairness and fidelity. Like committee-designed benchmarks (e.g., TPC-H, TPC-DS), a use-case-centric benchmark reflects its designer's view of a realistic workload. The distinction lies in *what* is fixed: committee benchmarks freeze the SQL text, a query set that vendors optimize against for years, whereas a use-case-centric benchmark fixes the data sources, tooling, and workflow, and lets the query text emerge from tools such as dbt. This decoupling preserves the fairness of a fixed specification while inheriting the fidelity of real engineering practice: CTAS-heavy pipelines, dashboard idioms, and string-heavy joins appear because the tools produce them, not because a committee chose them.
Toward systematic derivation. The prototype described above remains limited: it covers only two workload classes (ELT and dashboard queries) over a single synthetic dataset. Scaling beyond this requires two methodological choices: *coverage* across tooling stacks and data models to avoid overfitting, and *selection* of representative use cases spanning domains (e.g., ELT, ad hoc analytics). We view these as open design questions and sketch directions below.
Additional, heterogeneous data sources. A key challenge in ELT is integrating and harmonizing heterogeneous data sources [38]. An important next step is therefore to incorporate additional datasets and source formats (e.g., CSV, JSON, Parquet). For example, market surveys [25] and sales data from related industries (e.g., coffee shops [10]) are available from public repositories such as Kaggle. These datasets often require normalization and missing-value imputation, making them more realistic benchmark inputs [29].
Multiple use cases. To capture the multi-tenancy and diversity of modern cloud analytics platforms, the benchmark should include use cases drawn from different domains, such as financial transactions [2] and fantasy sports [19]. Composing such use cases manually is itself a bottleneck. Inspired by ELTBench [11], which evaluates agents on their ability to construct ELT pipelines, we envision use cases being expanded automatically by agents, not only for ELT, but also for downstream analysis tasks such as data exploration and dashboard queries, via APIs such as Apache Superset's. This future-proofs the benchmark for settings where a substantial fraction of SQL queries is generated by agents [16].

5 Conclusion

Cloud telemetry traces have given the database community valuable insight into production workloads and inspired a new class of synthesis-based benchmarks. Our results show that telemetry, in its current form, is insufficient for faithful query synthesis: it depends on the execution system and context and does not uniquely determine query structure. Consequently, telemetry-based synthesis can reproduce coarse-grained workload statistics while still failing to preserve the distinctive query-level properties of modern analytics workloads. We see two complementary paths forward: a call to industry to expose richer, engine-agnostic workload features beyond raw telemetry, and, for the research community, building benchmark workloads directly from open application scenarios such as ELT pipelines and BI dashboards. Together, they point toward benchmarks that are both realistic and portable across systems.

Acknowledgments

Funded/Co-funded by the European Union (ERC, CODAC, 101041375 and ERC, FDS, 101164556). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] 2026. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io. Accessed: 2026-04-24.
[2] Google BigQuery. 2026. Bitcoin Blockchain Historical Data — kaggle.com. https://www.kaggle.com/datasets/bigquery/bitcoin-blockchain. [Accessed 21-03-2026].
[3] dbt labs. [n. d.]. GitHub - dbt-labs/jaffle-shop: An open source sandbox project exploring dbt workflows via a fictional sandwich shop's data. — github.com. https://github.com/dbt-labs/jaffle-shop. [Accessed 19-03-2026].
[4] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeffrey F. Naughton, and Stratis Viglas. 2022. DIAMETRICS: benchmarking query engines at scale. *Commun. ACM* 65, 12 (2022), 105–112.
[5] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.
[6] Dominik Durner, Lennart Espe, Jana Giceva, and Anja Gruenheid. 2024. TracEx: Understanding and Analyzing Database Traces. In *CIDR*. www.cidrdb.org.
[7] Pascal Ginter and Viktor Leis. 2026. Active Data Lakes: Regaining Physical Data Independence Without Losing Interoperability. *Proc. VLDB Endow.* (2026).
[8] Tobias Götz, Daniel Ritter, and Jana Giceva. 2025. LakeVilla: A Modular and Non-Invasive Toolbox for Lakehouse Transactions. arXiv:2504.20768 [cs.DB] https://arxiv.org/abs/2504.20768
[9] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374.
[10] Yaroslav Isaienkov. 2025. Coffee Sales. doi:10.34740/KAGGLE/DSV/11159944
[11] Tengjun Jin, Yuxuan Zhu, and Daniel Kang. 2025. ELT-Bench: An End-to-End Benchmark for Evaluating AI Agents on ELT Pipelines. *Proc. VLDB Endow.* 19, 2 (2025), 84–98.
[12] Skander Krid, Mihail Stoian, and Andreas Kipf. 2025. Redbench: A Benchmark Reflecting Real Workloads. *CoRR* abs/2506.12488 (2025).
[13] Jiale Lao and Immanuel Trummer. 2025. Demonstrating SQLBarber: Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. In *SIGMOD Conference Companion*. ACM, 151–154.
[14] Jiale Lao and Immanuel Trummer. 2025. SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. *CoRR* abs/2507.06192 (2025).
[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
[16] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph Gonzalez, and Aditya G. Parameswaran. 2026. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. In *CIDR*. www.cidrdb.org.
[17] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *IEEE BigData*. IEEE Computer Society, 2884–2892.
[18] Michail Georgoulakis Misegiannis, Daniel Ritter, Viktor Leis, and Jana Giceva. 2025. CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics. *Proc. VLDB Endow.* 18, 13 (2025), 5638–5651.
[19] NFL. 2026. NFL Big Data Bowl 2026 - Prediction — kaggle.com. https://www.kaggle.com/competitions/nfl-big-data-bowl-2026-prediction/data. [Accessed 21-03-2026].
[20] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *CDMS@VLDB*.
[21] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30.
[22] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *SIGMOD Conference Companion*. ACM, 347–359.
[23] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.
[24] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744.
[25] Vijaya Shree Raja Sekaran. 2019. Food Preferences — kaggle.com. https://www.kaggle.com/datasets/vijayashreer/food-preferences. [Accessed 21-03-2026].
[26] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2015. SQLsmith. https://github.com/anse1/sqlsmith. Accessed: 2026-03-12.
[27] Till Steinert, Maximilian Kuschewski, and Viktor Leis. 2026. Cloudspecs: Cloud Hardware Evolution Through the Looking Glass. In *CIDR*. www.cidrdb.org.
[28] Murray Stokely, Neel Nadgir, Jack Peele, and Orestis Kostakis. 2025. Shaved Ice: Optimal Compute Resource Commitments for Dynamic Multi-Cloud Workloads. In *ICPE*. 124–135.
[29] Michael Stonebraker. 2014. Why the "Data Lake" is Really a "Data Swamp". BLOG@CACM. [Accessed 21-03-2026].
[30] Jan Vincent Szlang, Sebastian Breß, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights From the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proc. VLDB Endow.* 18, 12 (2025), 5126–5138.
[31] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil 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. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706.
[32] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.
[33] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.
[34] Chengcheng Wan, Yiwen Zhu, Joyce Cahoon, Wenjing Wang, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Alexandra M. Ciortea, Konstantinos Karanasos, and Subru Krishnan. 2023. Stitcher: Learned Workload Synthesis from Historical Performance Footprints. In *EDBT*. OpenProceedings.org, 417–423.
[35] Zhengle Wang, Yanfei Zhang, and Chunwei Liu. 2026. ResQ: Realistic Performance-Aware Query Generation. *arXiv preprint arXiv:2602.02999* (2026).
[36] 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. *CoRR* abs/2511.13059 (2025).
[37] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison W. Lee. 2018. Snowtrail: Testing with Production Queries on a Cloud Database. In *DBTest@SIGMOD*. ACM, 4:1–4:6.
[38] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. www.cidrdb.org.
[39] Yan Zhou, Chunwei Liu, Bhuvan Urgaonkar, Zhengle Wang, Magnus Mueller, Chao Zhang, Songyue Zhang, Pascal Pfeil, Dominik Horn, Zhengchun Liu, Davide Pagano, Tim Kraska, Samuel Madden, and Ju Fan. 2025. PBench: Workload Synthesizer with Real Statistics for Cloud Analytics Benchmarking. *Proc. VLDB Endow.* 18, 11 (2025), 3883–3895.

A SQL Structural Features

Table 1 lists the 27 structural features extracted from each SQL query via its `sqlglot` abstract syntax tree (AST). These features form the basis of the similarity analysis in Section 3.3.

B Example Burner Query

ResQ injects artificial compute expressions to meet the telemetry target of a given workload trace (Section 3.3). The query below targets a three-table join between `nation`, `supplier`, and `partsupp`; it selects every column from all three tables, groups by all of them, and appends `SIN()` and `MD5()` expressions to hit the target CPU time. The `SIN`/`MD5` aggregates add compute cycles but do not correspond to typical analytical aggregations, illustrating how telemetry-only optimization can produce queries that match the target metric without reflecting real workload patterns.

```
SELECT
  nation.n_comment, nation.n_nationkey,
  nation.n_name, nation.n_regionkey,
  partsupp.ps_partkey,
  supplier.s_comment, supplier.s_suppkey,
  supplier.s_nationkey, supplier.s_acctbal,
  supplier.s_phone,
  MD5(nation.n_name) AS n_name,
  MD5(supplier.s_comment) AS s_comment,
  MD5(supplier.s_phone) AS s_phone,
  SIN(nation.n_nationkey) AS n_nationkey,
  SIN(nation.n_regionkey) AS n_regionkey,
  SIN(partsupp.ps_partkey) AS ps_partkey,
  SIN(supplier.s_suppkey) AS s_suppkey,
  SIN(supplier.s_nationkey) AS s_nationkey,
  SIN(supplier.s_acctbal) AS s_acctbal,
  COUNT(*) AS row_count
```

Table 1: SQL structural features used for query similarity analysis.

#	Feature	Description
Count features		
1	n_tables	Distinct tables referenced
2	n_joins	JOIN clauses
3	n_filters	WHERE clauses
4	n_aggregations	Aggregate calls (SUM, AVG, COUNT, MIN, MAX)
5	n_group_by_cols	Columns in GROUP BY
6	n_order_by_cols	Columns in ORDER BY
7	n_select_cols	Expressions in SELECT
8	n_subqueries	Nested subqueries
9	n_and	AND connectives
10	n_or	OR connectives
11	n_comparison_ops	Comparison operators (=, >, <, ≥, ≤, ≠)
12	n_math_funcs	Math functions (sin, cos, abs, round, md5, ...)
13	n_string_funcs	String functions (substr, upper, trim, concat, ...)
14	n_date_funcs	Date functions (dateadd, extract, year, ...)
Boolean features		
15	has_having	Contains HAVING clause
16	has_limit	Contains LIMIT clause
17	has_distinct	Contains DISTINCT
18	has_union	Contains UNION
19	has_case	Contains CASE expression
20	has_window	Contains window function
21	has_exists	Contains EXISTS predicate
22	has_in_subquery	Contains IN predicate
23	has_cte	Contains common table expression (WITH)
24	has_between	Contains BETWEEN predicate
25	has_like	Contains LIKE predicate
26	has_not	Contains NOT operator
Structural feature		
27	depth	Maximum AST nesting depth

```
FROM nation
JOIN supplier
  ON nation.n_nationkey = supplier.s_nationkey
JOIN partsupp
  ON supplier.s_suppkey = partsupp.ps_partkey
GROUP BY
  nation.n_comment, nation.n_nationkey,
  nation.n_name, nation.n_regionkey,
  partsupp.ps_partkey,
  supplier.s_comment, supplier.s_suppkey,
  supplier.s_nationkey, supplier.s_acctbal,
  supplier.s_phone
```

Received 20 March 2026; accepted 19 April 2026