Boosting DBMS Test Coverage via LLM-Driven SQL Generation

Eslam Abdelkarim
TU Darmstadt
Darmstadt, Germany

Carsten Binnig
TU Darmstadt & hessian.AI & DFKI
Darmstadt, Germany

Anupam Sanghi
IIT Hyderabad
Sangareddy, India

Abstract

Database management systems require comprehensive testing, but achieving high code coverage in complex DBMS implementations remains challenging. Manual test writing is time-consuming, while automated query generation tools struggle with schema flexibility and coverage guidance. This paper presents Quover, an automated approach for improving DBMS test coverage through large language model (LLM)-generated SQL queries. Specifically, Quover implements an iterative coverage-guided methodology that identifies uncovered functions in the target DBMS source code and generates targeted SQL queries. In each iteration, an LLM call is made with a rich context, including function descriptions and code snippets, and their effectiveness is validated. Our experiments show that over 57% coverage can be achieved within a few hours–already representing a 14% improvement over state-of-the-art automated systems. Furthermore, we evaluate Quover across multiple database schemas and DBMSs, demonstrating its effectiveness as an automated approach alongside handcrafted test suites.

CCS Concepts

• **Information systems** → **Database utilities and tools**; **Database management system engines**; **Structured Query Language**.

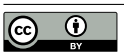
Keywords

SQL generation; Database coverage testing

ACM Reference Format:
Eslam Abdelkarim, Carsten Binnig, and Anupam Sanghi. 2026. Boosting DBMS Test Coverage via LLM-Driven SQL Generation. In *Testing Database Systems (DBTest '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3810991.3811637

1 Introduction

Coverage Testing of Database Systems is Critical. As DBMS implementations grow more complex, with increasingly rich SQL dialects and sophisticated query optimizers and executors, it becomes harder to reason about how well they have been exercised by existing test suites. Coverage-based evaluation provides a systematic way to measure how much of the implementation is actually tested, quantifying the extent to which different query plans and execution paths are explored. This helps identify which planner decisions and executor code paths have been validated under realistic workloads and which remain untested.



DBTest '26, Bengaluru, India

ACM ISBN 979-8-4007-2701-6/2026/05
https://doi.org/10.1145/3810991.3811637

Current Landscape of Database Testing. Today, handcrafted test suites typically developed by DBMS engineers and the open-source community play a central role in validating correctness and stability across a wide range of features. For example, the PostgreSQL regression tests [2] provide a comprehensive set of SQL tests for the system's core functionality, reaching over 80% line coverage. Standard benchmark suites such as TPC-H [9] and TPC-DS [10] provide widely adopted, carefully designed workloads for evaluating and comparing DBMS performance under realistic decision-support scenarios. In addition to such manually curated suites, a variety of automated approaches have been proposed for generating and executing SQL workloads. SQL fuzzing tools such as SQLancer [3, 6] automatically generate syntactically valid queries and use logic-testing oracles (e.g., Pivoted Query Synthesis) to uncover logic bugs in DBMSs. Coverage-guided fuzzers such as RATEL [11] and SQLRight [1] incorporate code coverage feedback to iteratively refine and prioritize SQL inputs, facilitating effective exploration of diverse execution paths in database systems. More recently, LLM-based systems such as SQLStorm [8] leverage semantic understanding to generate diverse, realistic SQL workloads at scale, showing that LLMs can efficiently synthesize large corpora of queries that reflect real-world patterns.

Existing Approaches are not Enough. While the aforementioned approaches are useful, they face the following challenges.

(a) Lack of Focus on Improving Coverage: Benchmarks typically exercise only a limited subset of internal components, while fuzzing-based generators lack semantic guidance to target specific features or functions. Moreover, existing LLM-based systems, such as SQLStorm, do not leverage coverage feedback to systematically explore and exercise uncovered code paths.

(b) Lack of Automation: Handcrafted tests that target uncovered parts incur significant manual overhead, rendering them impractical for large-scale or customer-specific deployments. Automation is therefore essential to ensure system robustness across diverse client environments.

Quover Addresses the Above Lacunae. In this paper, we present Quover, an automated system that addresses the above challenges for custom schemas. Specifically, it closes the coverage feedback loop by combining LLM-driven SQL generation with fine-grained coverage measurement to systematically target uncovered DBMS code paths. Quover operates in an iterative loop: after each iteration, we measure coverage, identify uncovered functions and lines of code, and construct a structured context that is sent to an LLM to generate new targeted queries. Specifically, the context includes coverage data, function descriptions, source code snippets, schema information, and examples of queries that previously covered similar code. The generated queries are executed, and execution feedback and redundancy metrics are used to refine prompts, while plateau detection signals when coverage stalls for later analysis.

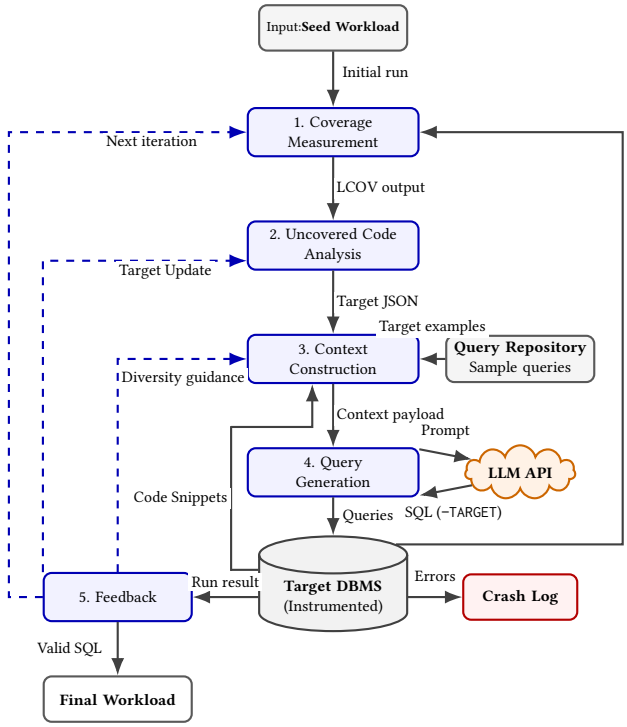


Figure 1: Quover's five-stage pipeline: Each iteration measures coverage (§2.1), selects uncovered target functions (§2.2), constructs an LLM context (§2.3), generates targeted SQL queries (§2.4), and provides feedback for next iteration (§2.5).

Initial Results. In this paper, we evaluate Quover on PostgreSQL and DuckDB systems, using the schemas from TPC-H and TPC-DS benchmarks as targets, and demonstrate consistent coverage improvements across all the settings. More concretely, our automated approach routinely pushes line coverage from base levels of roughly 20–40% into the 50–70% range. This rapid exploration achieves up to a 14 percentage point improvement over a prior automated system within just a few hours.

2 System Architecture

The architecture of Quover, as shown in Figure 1, comprises of an iterative pipeline of 5 stages, where stages (1)–(2) produce a coverage report by executing queries and identify high priority target snippets in the source code; stages (3)-(4) curate the context and generate new queries; the final stage (5) assesses the result and provides feedback for the next iteration.

2.1 Coverage Measurement

Instrumentation. The target DBMS is compiled with GCC's coverage flags to measure line and branch coverage during query execution, and coverage is scoped to a single component of the target DBMS per run (for example, an executor or optimizer module). Since LCOV counters are cumulative, coverage from all previous iterations is preserved automatically.

2.2 Uncovered Code Analysis

Function-level analysis. The LCOV output is parsed into function-level statistics: for every function in the scoped component, we extract its name, file path, line ranges, current coverage percentage, and a description derived from source code comments.
Priority ranking. Functions are ranked using a weighted score: $score = uncoveredCount \times \frac{\max(1, currentIter - lastIteration)}{timesSelected+1}$. Here, $uncoveredCount$ prioritizes functions with more uncovered lines, the $\max(1, currentIter - lastIteration)$ term boosts functions not selected recently, and $(timesSelected + 1)$ down-weights frequently selected ones. This favors functions with many uncovered lines that have not been selected recently. The top F functions are chosen as high-priority targets for the iteration.
Structured output. The analysis produces a JSON document with a coverage_summary (overall statistics), high_priority_targets (files and their uncovered functions with line ranges), and the database schema. This JSON is forwarded to the next stage.

2.3 Context Construction

The central contribution of our approach is the construction of a structured context payload that guides the LLM toward generating queries likely to exercise specific uncovered code paths. The payload is assembled by progressively injecting several types of information.
Base payload. A *system message* establishes the LLM's role as a database testing expert and identifies the target DBMS and component. A *main prompt* provides detailed query-generation instructions. The coverage JSON and the component's architectural documentation (if available) are appended.
Code snippets. For the functions with the largest uncovered ranges, the source code (up to 30 lines per range, for up to 6 functions) is included so the LLM can reason about what conditions or inputs are needed to reach the uncovered lines.
Target query examples. To support this step, the system maintains a repository of existing workloads, which can either consist of the built-in regression test suite of the target DBMS or an external workload imported from another database. From this repository, the system can optionally maintain a mapping from existing queries to the specific lines they cover. A *greedy set-cover* algorithm then selects example queries that maximize coverage of the currently uncovered lines, giving the LLM concrete patterns to build on.
Feedback. Starting from the second iteration, feedback from prior iterations is used to enrich the next prompt (details in Section 2.5). Note that each of the above context construction components is optional, and we show their impact on coverage in Section 3.3.

2.4 Query Generation

LLM invocation. The assembled message payload is sent to an LLM, which is instructed to produce exactly Q queries per iteration (a tunable parameter), each annotated with a `-- TARGET: <function_name>` comment indicating the intended uncovered function.
Expected-fail queries. Some uncovered code paths are only reachable through error-handling logic (e.g., constraint violations, type mismatches). The system explicitly instructs the LLM to generate queries annotated with `-- EXPECT_FAIL` to trigger these paths.

Execution. Each generated query is executed against the target DBMS subject to a configured statement timeout. Successful executions and expected failures remain in the workload and contribute to coverage, while timeouts and queries that cause the DBMS process to terminate abnormally are appended to the crash log. The systematic analysis of this log of potential execution anomalies is left for future work.

2.5 Feedback

After execution, several analyses feed back into the next iteration.
Coverage re-measurement. The current iteration's coverage is compared with the previous one to identify newly covered code, updating the uncovered-target pool for the next selection stage.
Query effectiveness analysis. Each query is mapped to the functions it actually covered, and compared against its target. Effective queries are promoted as positive examples in the next prompt.
Redundancy detection. Redundancy within an iteration and across iterations is measured; when found to be high, the next prompt adds stronger diversity guidance to steer exploration toward different functions.
Plateau detection. If coverage improvement stays below a threshold (0.1%) for three consecutive iterations, the system logs a plateau to mark where progress stalled for later analysis.
Termination. The process terminates when either the predefined coverage target (e.g., 80%) is achieved or the maximum number of iterations is reached.

3 Evaluation

In this section we present our experimental findings – specifically, we show the coverage improvement against a prior automated query generator. Subsequently, we discuss the generalizability of Quover across different systems and target databases. Finally, we present our ablation study to evaluate the contribution of different components of our solution.

3.1 Experimental Setup

DBMS configurations. We evaluate Quover on the executor and optimizer components of PostgreSQL 16 and DuckDB v1.2.1, previously instrumented with GCC/LCOV.
Seed database and queries. We use the TPC-H 1 GB [9] and the TPC-DS 1 GB [10] benchmarks to ground the schema and database. For each, the standard queries serve as the seed workload (i.e., the initial test suite), primarily to facilitate comparison and provide a consistent starting point. We emphasize that Quover does not depend on this seed; it requires only a target schema. The 1 GB scale factor is deliberate: large tables lead the optimizer to select richer execution strategies (e.g. hash joins, merge joins, and multi-phase aggregations) that are typically not exercised on small datasets.
LLM configuration. We use OpenAI's gpt-4o-mini with temperature 0.15, chosen for its cost-effectiveness — a 50-iteration run costs well under $1; evaluating stronger models is left for future work.
Hyperparameters. We run up to 50 iterations per configuration. Each iteration generates exactly $Q = 20$ queries and selects the top $F = 15$ functions as high-priority targets. Based on our experiments, we found that $Q = 20$ and $F = 15$ work well; the number of example queries provided to the LLM matches Q (20 queries).

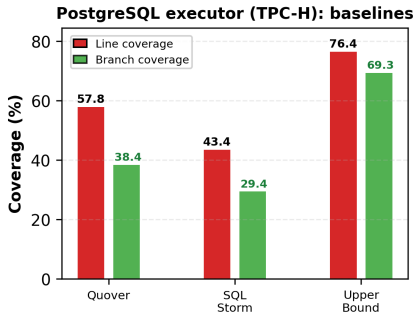


Figure 2: PostgreSQL executor TPC-H coverage: grouped bars show Quover, SQLStorm (averaged over 20×1k-query samples), and a *TPC-H-schema upper bound.*

Baselines and upper bound. We compare Quover against SQLStorm [8], an LLM-based workload generator, using the authors' released query corpus and configuration. We chose SQLStorm because, like Quover, it uses an LLM to generate a diverse workload — making it the most directly comparable automated approach.

SQLStorm results report *mean* line and branch coverage over 20 independent draws of 1,000 queries each (20×1k). For Quover, we measure coverage after executing up to 1,000 LLM-generated queries produced over 50 iterations × 20 queries per iteration.

To contextualize these numbers, we also estimate an *upper bound* on achievable coverage by excluding from the line/branch denominators the code in functions that cannot be exercised under our chosen database. Concretely, we manually inspect each function's source-level documentation and flag those that reference features absent from the TPC-H/TPC-DS schema (e.g. partitioned tables, materialized views, parallel replication, foreign-data wrappers), or that are explicitly marked obsolete or deprecated. This bound is intentionally conservative: additional internal paths still in the denominator are guarded by specific table types, data distributions, or DBMS configurations that the fixed schema never provides, and reaching them requires *mutating* the schema and DBMS configuration. We plan to address this as part of future work.

3.2 Coverage Improvement

Figure 2 presents PostgreSQL *executor* TPC-H coverage with paired line/branch bars for Quover, SQLStorm, and a TPC-H-schema upper bound. In this plot, Quover improves over SQLStorm on both executor line coverage (57.83% vs. 43.42%) and executor branch coverage (38.39% vs. 29.4%), while remaining below the TPC-H-schema upper bound (76.40% line / 69.29% branch).
Cost and runtime. On our hardware, a 50-iteration run of Quover completed in roughly 3 hours on average, with LLM API costs well below $1 per run when using `gpt-4o-mini`. For comparison, SQLStorm runs 7 batches in its main generation phase, and reports that each batch of 5,000 `gpt-4o-mini` requests is processed within one day (typically faster) and costs $2 on average [8].
Query validity. Across the 50-iteration PostgreSQL executor run, nearly 20% of the LLM-generated queries fail due to schema mismatches, unsupported constructs, or timeouts; these are added to the crash log.

Table 1: Line/branch coverage across different systems (PostgreSQL and DuckDB) and databases (TPC-H and TPC-DS).

	TPC-H		TPC-DS	
Config.	**Line**	**Branch**	**Line**	**Branch**
PG executor	57.83%	38.39%	55.81%	38.10%
PG optimizer	62.31%	48.34%	64.89%	50.75%
DuckDB executor	51.12%	16.16%	54.64%	18.41%
DuckDB optimizer	64.56%	35.38%	68.58%	28.29%

Generalizability. Table 1 reports line and branch coverage for PostgreSQL and DuckDB on TPC-H and TPC-DS seed databases, and across all four configurations. Quover consistently boosts both metrics over the seed workloads, delivering approximately 20–30% higher line coverage and 10–20% higher branch coverage for both systems across all configurations. This indicates that our coverage-guided generation generalizes across different DBMSs and benchmark schemas. The table also shows that DuckDB's executor branch coverage remains lower than PostgreSQL's despite similar line coverage, as our TPC-H/TPC-DS runs use pre-loaded data and skip bulk import and CSV parsing. In DuckDB, this logic resides within the executor, so many import-related branches remain unexecuted and are not counted in coverage, whereas PostgreSQL isolates such code in separate modules.

3.3 Ablation Study

In our ablation study on the PostgreSQL executor under the TPC-H schema, we use a leave-one-out design. Figure 3 shows line coverage over 50 iterations: the full system reaches 57.83%, while removing README documentation, uncovered query examples, code snippets, and feedback reduces coverage to 55.51%, 52.54%, 52.40%, and 51.80%, respectively; a prompt-only setup achieves 40.59% coverage. The iterative coverage-feedback loop itself drives a substantial share of the gain, where every configuration improves steadily over iterations. Overall, each component contributes to clear coverage gains.

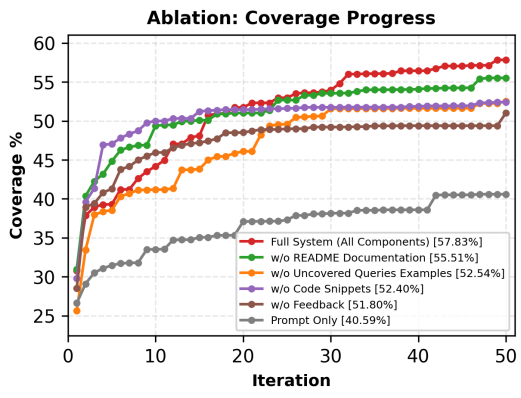


Figure 3: Line coverage progress over iterations for each ablation configuration on the PostgreSQL executor (TPC-H). The full system achieves the highest coverage, while removing individual components shows their relative contribution.

4 Related Work

Two active research directions are particularly relevant to our work: coverage-guided fuzzing for database systems and LLM-based SQL generation for testing. We briefly survey representative systems in each area and position Quover relative to them.

Fuzzing-based coverage targeting. Logic-testing tools such as SQLancer [3, 6] generate queries and pair them with metamorphic or partitioning-based oracles (e.g., Pivoted Query Synthesis (PQS) [6], NoREC [4], and Ternary Logic Partitioning (TLP) [5]) to uncover logic bugs and exercise diverse execution paths in DBMSs, but they do not explicitly optimize for exercising uncovered code regions. Coverage-guided fuzzers such as SQLRight [1], RATEL [11], and SQLaser [12] steer SQL generation using code coverage feedback and validity oracles, increasing bug-finding effectiveness over pure differential testing. Across these systems, coverage is used as a mutation signal: they lack semantic understanding of what an uncovered function computes and therefore cannot construct queries that target specific internal code paths. Quover instead provides the LLM with function descriptions and code snippets to enable semantics-aware targeting of uncovered regions.

LLM-based coverage targeting. Several systems leverage Large Language Models (LLMs) to generate SQL workloads and tests. SQLStorm [8] uses GPT-based models and carefully engineered prompts to synthesize diverse analytical SQL queries for multiple DBMSs, focusing on workload diversity and realistic benchmarking rather than explicit coverage objectives. ShQveL [13] integrates LLM-generated SQL fragments into an existing generator via "SQL sketching", validating successful fragments and reusing them to increase feature coverage and uncover new DBMS bugs. Beyond databases, LLM-based test and data generation [7] has been shown to improve coverage in general software testing.Quover combines fine-grained coverage feedback with LLM-driven SQL generation in a closed loop, explicitly targeting uncovered DBMS functions without relying on external query generators.

5 Conclusion

This paper presents Quover, an automated approach for improving DBMS test coverage through LLM-generated SQL queries. Quover demonstrates practical effectiveness, achieving coverage gains on PostgreSQL and DuckDB. The iterative, coverage-guided methodology successfully leverages LLMs to generate targeted queries that exercise uncovered code paths, providing a promising alternative to manual test writing.

As part of future work, we plan to extend Quover with *database-invasive* mutation of the schema, data distribution, and DBMS configuration to unlock code paths beyond what query variation alone can reach. Further, we plan to leverage the generated workloads and the crash log to systematically discover bugs in the target system.

Acknowledgments

This work was supported by the LOEWE Spitzenprofessur (III 5-519/05.00.003-(0005)), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC-3057/1 "Reasonable Artificial Intelligence", Project No. 533677015). We also thank DFKI and hessian.AI for their support.

References

[1] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association, 4309–4326.

[2] PostgreSQL Global Development Group. 2026. PostgreSQL Documentation: Regression Tests. https://www.postgresql.org/docs/current/regress.html.

[3] Manuel Rigger. 2022. SQLancer. https://github.com/sqlancer/sqlancer. Accessed 2026-02-25.

[4] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, 1140–1152. doi:10.1145/3368089.3409710

[5] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020), 30 pages. doi:10.1145/3428279

[6] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, Article 38, 16 pages.

[7] Sujoy Roy Chowdhury, Giriprasad Sridhara, A K Raghavan, Joy Bose, Sourav Mazumdar, Hamender Singh, Srinivasan Bajji Sugumaran, and Ricardo Britto. 2025. Static Program Analysis Guided LLM Based Unit Test Generation. In *Proceedings of the 8th International Conference on Data Science and Management of Data (CODS-COMAD '24)*. ACM, 279–283. doi:10.1145/3703323.3703742

[8] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *PVLDB* 18, 11 (2025), 4144–4157. doi:10.14778/3749646.3749683

[9] Transaction Processing Performance Council. 2022. TPC Benchmark^{TM} H Standard Specification. https://www.tpc.org/tpch/ Version 3.0.1.

[10] Transaction Processing Performance Council. 2024. TPC Benchmark^{TM} DS Standard Specification. https://www.tpc.org/tpcds/ Version 4.0.0.

[11] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '21)*. IEEE Press, 328–337. doi:10.1109/ICSE-SEIP52600.2021.00042

[12] Jin Wei, Ping Chen, Kangjie Lu, Jun Dai, and Xiaoyan Sun. 2026. SQLaser: Detecting database management system (DBMS) logic bugs with clause-guided fuzzing. *Journal of Computer Security* 34, 1 (2026), 3–28. doi:10.1177/0926227X251370258

[13] Suyang Zhong and Manuel Rigger. 2025. Testing Database Systems with Large Language Model Synthesized Fragments. arXiv:2505.02012 [cs.SE]