Testing Computation Pushdown in Distributed Database Systems

JINSHENG BA, ETH Zurich, Switzerland
ZUMING JIANG, University of Hong Kong, Hong Kong
ZHENDONG SU, ETH Zurich, Switzerland

Computation pushdown is a critical technique in distributed database management systems (DBMSs), enabling certain operations to be executed closer to the data to reduce network overhead and improve performance. However, its behavior depends on multiple factors beyond the input query itself, such as data distribution and resource utilization. This makes it difficult to validate correctness using only input queries in a black-box manner. Existing testing methods that rely solely on query manipulation cannot effectively control or predict pushdown behavior, and are therefore insufficient. In this paper, we introduce *Controlled Pushdown Execution (CPE)*, a white-box method that enables systematic validation of computation pushdown. CPE modifies the source code of DBMSs to forbid a specific pushdown operator and compares the results. Any discrepancy reveals a bug. Our study shows that CPE can control all supported operators across different systems. We applied CPE to three production-grade distributed DBMSs: CockroachDB, TiDB, and YugabyteDB. CPE found 25 previously unknown and unique bugs, 14 of which are logic bugs—incorrect results. CPE finds 3× more bugs than historical bugs and can reproduce all historical bugs. Beyond computation pushdown, the core insight of controllable execution can generalize to other contexts (*e.g.*, transaction schedule), providing a systematic way to uncover subtle logic bugs.

CCS Concepts: • **Information systems → Relational parallel and distributed DBMSs**; • **Software and its engineering → Software testing and debugging**.

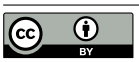
Additional Key Words and Phrases: Distributed database systems, logic bug, computation pushdown

ACM Reference Format:
Jinsheng Ba, Zuming Jiang, and Zhendong Su. 2026. Testing Computation Pushdown in Distributed Database Systems. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA001 (October 2026), 23 pages. https://doi.org/10.1145/3832092

1 Introduction

Database Management Systems (DBMSs) are critical and widely used systems to store and retrieve data [17, 31, 38]. Unlike centralized DBMSs, which process all data in a single machine, distributed DBMSs distribute both storage and computation across multiple nodes to accommodate growing data volumes and user demands [23]. In such systems, processing an input query requires loading data from various nodes to a single node before computing the result. This approach incurs substantial performance overhead due to the high cost of network data transmission. To address this, a critical technique in distributed DBMSs is *computation pushdown*, which decomposes and pushes computation as close to the data storage nodes as possible, minimizing network traffic and improving performance. Pushdown is thus a fundamental distinction between centralized

Authors' Contact Information: Jinsheng Ba, ETH Zurich, Zurich, Switzerland, jinsheng.ba@inf.ethz.ch; Zuming Jiang, University of Hong Kong, Hong Kong, Hong Kong, jzuming@hku.hk; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.



ACM 2994-970X/2026/10-ARTISSTA001
https://doi.org/10.1145/3832092

and distributed DBMSs, and its correctness is mission-critical to ensure reliable query results in distributed environments.

Automatically finding logic bugs (*i.e.*, incorrect results) in the computation pushdown requires a reliable test oracle, which validates the result. Designing such a test oracle faces two key challenges. First, computation pushdown lacks any standardized specification, giving implementers wide latitude. As a result, pushdown behavior is typically based on developer intuition, with no definitive ground truth to serve as a reference. Second, the execution of computation pushdown cannot be reliably decided by inputs alone. Given an input query, distributed DBMSs deploy an internal scheduler to determine where and how to do the computation. The schedule is affected by multiple factors, including data distribution and resource utilization, which are not visible to black-box tests. An input alone provides insufficient information to predict or control the scheduler's choices, making it inefficient for examining execution behavior consistency, which is commonly used by differential testing (comparing results across systems) [4, 32], and metamorphic testing (comparing results of semantically relevant inputs) [13, 25, 26].

Existing methods show limitations in solving this problem. Most distributed systems testing, such as Jepsen [41], uses fault injection (*e.g.*, node crashes) to validate consistency under failures [19]. While effective for testing fault-tolerance mechanisms, Jepsen cannot detect logic bugs in computation pushdown, which occur during normal execution due to incorrect operator implementations or scheduling decisions. Some test oracles target logic bugs in centralized DBMSs by manipulating inputs and checking the consistency of execution results. Techniques such as *Ternary Logic Partitioning* (TLP) [26] and *Equivalent Expression Transformation* (EET) [13] generate pairs of semantically equivalent queries and compare their outputs. These methods are effective for validating the correctness of specific features—such as `WHERE` clauses—whose execution is typically determined solely by inputs in centralized settings. However, the execution of computation pushdown is determined by factors like data distribution, which is independent of the input queries. Therefore, both cannot efficiently validate the implementation of computation pushdown. A potential technique is differential testing. For example, one could compare the results of a distributed DBMS running in a single-node deployment versus a multi-node deployment. Yet, computation pushdown is often unaffected by the number of nodes [49], as the system may still push down computation even in a single-node configuration if the storage and execution layers are decoupled. This limits the effectiveness of such an approach in exposing computation pushdown bugs.

In this paper, we propose a novel and practical white-box test oracle to find logic bugs in computation pushdown. Our study of real-world distributed DBMSs reveals that, while each DBMS supports a customized set of pushdown operators, they share a common mechanism: pushdown decisions are made by scheduling functions that check conditions such as query semantics and data distribution. If these conditions are met, computation is pushed down to the storage layer; otherwise, the DBMS loads all data into a single node before computation. Our key idea is to disable a specific pushdown operator by modifying the source code, and then validating its correctness by comparing execution results with and without pushdown. To this end, CPE makes pushdown execution both controllable and checkable, addressing the two core challenges in designing a reliable test oracle. To identify the functions for pushdown decisions. We use static analysis to locate candidate functions and then validate them using the system's official test suite and Large Language Models (LLMs) [5, 50], which are assumed to provide comprehensive coverage.

Listing 1 shows a motivating example bug we found in the distributed DBMS YugabyteDB. Consider an e-commerce platform that stores its product inventory in YugabyteDB. Lines 1–3 create the `orders` table, define an index, and insert sample data representing order history. A daily analytics job then runs the query in line 4 to check whether any product was sold before a given date for inventory forecasting. If the condition is satisfied, the query should return the

Listing 1. A bug we found in computation pushdown.

```
1  CREATE TABLE orders(order_id INT, order_date DATE);
2  CREATE INDEX i0 ON orders(order_date, order_date ASC);
3  INSERT INTO orders VALUES(1, '2025-08-20'), (2, '2025-08-21'), (3,
       '2025-08-25');
4  SELECT DISTINCT 'exist' FROM orders WHERE orders.order_date<'2025-09-01';
5           -- {exist} ✔without pushdown, {} 🐞 with pushdown
6  -------------------------------Code Manipulation-------------------------------
7  src/postgres/src/backend/optimizer/path/indxpath.c
8   -4757,6 +4757,7  yb_can_pushdown_distinct(...) {
9  --        return true;  // with pushdown
10 ++        return false; // without pushdown
```

string "exist"; otherwise, it should return an empty result. In this case, the query in line 4 should return "exist" because the expression orders.order_date<'2025-09-01' is true. Due to a bug in the computation pushdown of **DISTINCT**, the result is incorrect. As shown in lines 7–10, CPE identified the function can_pushdown_distinct(), which decides whether to push down the execution of **DISTINCT**. CPE modifies this function to always return **false**, so that it provides a reference execution without pushdown to disclose this bug. EET and TLP fail to detect this bug because both test cases they generate are executed with **DISTINCT** pushdown enabled, resulting in no observable discrepancy. We reported this bug to developers, who quickly fixed it with around 200 lines of code, showing that it is a non-trivial bug. In a real-world setting, such a bug could lead the platform to mistakenly conclude that no products were sold, causing inventory underestimation, stockouts, and lost revenue.

We evaluated CPE in three widely used distributed DBMSs, CockroachDB, TiDB, and YugabyteDB. We found 25 unique and previously unknown bugs, in which 14 bugs are logic bugs, and 13 logic bugs are related to the computation pushdown. CPE is necessary for uncovering these bugs: existing testing methods TLP and EET failed to reproduce 12 logic bugs found by CPE, and 10 bugs had existed before the publication dates of TLP and EET. CPE is significant as it found 3× more bugs than historical bugs, and successfully reproduced all of them.

Overall, we make the following contributions:

- A study of pushdown in real-world distributed DBMSs.
- A novel white-box test oracle to identify logic bugs in computation pushdown.
- We implemented and evaluated the approach, which has found 14 unique, previously unknown logic bugs.

2 Background

Distributed DBMSs and computation pushdown. Figure 1 illustrates the architectures of centralized and distributed DBMSs, shown on the left and right sides, respectively. In both architectures, the SQL layer parses and processes a given query and retrieves the relevant data from the storage layer. Let us consider the query **SELECT MAX**(c0)**FROM** t0;. In the centralized architecture, the SQL layer retrieves all five rows of table t0 from the storage layer and then applies the aggregate function **MAX**(c0) to compute the result 99, as requested by the query. Since both the SQL and storage layers reside on the same machine, the data can be efficiently loaded into memory and processed locally before returning the result to the user. In contrast, in the distributed architecture, the SQL and storage layers typically reside on different machines. As a result, data must be transmitted over the network, which is significantly slower than memory access and constitutes a major performance bottleneck. To better utilize the computing resources of distributed systems and improve query execution performance, the SQL layer often decomposes and pushes down certain operators to the

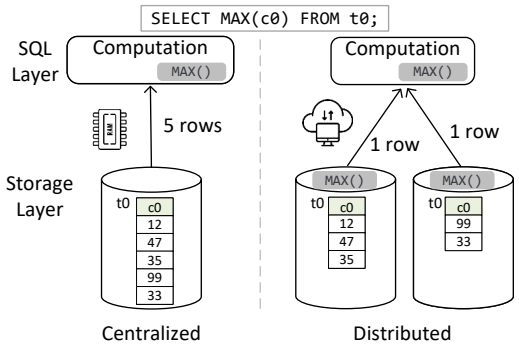


Fig. 1. Architectures of centralized and distributed DBMSs.

Table 1. Supported computation pushdown operators.

DBMS	Aggregate	Distinct	Expression	Function	GroupBy	Join	Topk	Union	Sum
CockroachDB	✓	✓	✓	✓	✓	✓	✓	✓	8
TiDB	✓	✓	✓	✓	✓	✓	✓	✓	8
YugabyteDB	✓	✓	✓	✓	✗	✗	✗	✗	4

storage layer. In this example, the SQL layer decomposes and pushes the `MAX()` aggregate function down to the storage layer, allowing each node to compute a local maximum. Consequently, only one row per node is sent over the network, significantly reducing data transfer and improving performance. Computation pushdown is one of the most distinctive features of distributed DBMSs compared to centralized DBMSs, and validating its correctness is important to ensure correct results in distributed settings. In this work, we propose a novel white-box method to identify logic bugs in computation pushdown.

3 Computation Pushdown Study

To facilitate our testing method, we characterized how computation pushdown is supported and controlled across DBMSs. Each DBMS implements pushdown empirically, often with a custom set of supported operators and scheduling logic. This lack of transparency makes it unclear what can or cannot be pushed down, or how these decisions are made, posing a major challenge for systematic testing. To build a test oracle, we must understand the extent of pushdown support and the internal mechanisms behind it. Thus, we conducted a systematic study of real-world distributed DBMSs to answer the following two key research questions.

RQ1 Supported Operators. What pushdown operators are supported in real-world DBMSs?
RQ2 Scheduling Logic. What is the general logic pattern for scheduling operators?

Study targets. We studied three widely used distributed DBMSs: CockroachDB, TiDB, and YugabyteDB. CockroachDB and TiDB are native distributed systems written in Go, with their open-source repositories on GitHub receiving over 30.9k and 38.5k stars, respectively. YugabyteDB is a distributed version of PostgreSQL, one of the most widely used centralized DBMSs. It reuses a significant portion of PostgreSQL's codebase to provide compatibility while enabling a distributed architecture. These DBMSs have been extensively evaluated in prior research works [2, 3, 15, 26]. We evaluated the latest development versions of each DBMS: CockroachDB (7625d9f), TiDB (b1a5536), and YugabyteDB (2003789).

Identifying computation pushdown operators. We followed the prior research [48] and examined DBMS documentation to identify the potential SQL operators supported for computation pushdown. Not all SQL operators can be decomposed and pushed down to the storage layer. To identify candidate pushdown operators, we searched for the keyword "pushdown" in the documentation of DBMSs and cross-referenced the findings with previous work [48].

Validating pushdown operators. For each candidate pushdown operator, we designed a benchmark to validate its support. The benchmark uses a table named `employees`, which contains two integer fields: `id` and `salary`. We constructed at least one query for each operator and examined its query plans to see whether the corresponding operator is enabled. Query plans—trees of operations that describe how SQL statements are executed by a specific DBMS—serve as the basis for this validation. In CockroachDB, query plans include a `distribution` property: **`local`** indicates that pushdown is not applied, while **`full`** indicates that it is. In TiDB, each operation in the query plan specifies its execution location. For instance, `cop` denotes execution in the storage layer, whereas `root` indicates execution in the SQL layer. YugabyteDB does not provide explicit indicators for pushdown in its query plans. Instead, YugabyteDB shows the number of network communications in query plans, such as **`Storage Table Read Request`**. When pushdown is enabled, network communication is typically minimal—usually just once. Additionally, certain operations—such as **`Storage Filter`**—may appear in query plans when pushdown is applied. Therefore, we also checked for the presence of such operations to determine.

RQ1 Supported Operators

Presentation. Table 1 shows the supported computation pushdown operators in CockroachDB, TiDB, and YugabyteDB. *Aggregate* denotes scalar aggregation operators, such **`COUNT`**`()` and **`MIN`**`()`. Executing them in the storage layer transfers only results, instead of all rows, over the network. *Distinct* denotes the operators to duplicate rows, aiming to reduce the number of rows transferred over the network. *Expression* denotes the expressions to evaluate data. The query we used is **`SELECT`** `*` **`FROM`** `employees` **`WHERE`** `salary = 2000`, which transfers only the rows that satisfy `salary = 2000`, instead of all rows. *Function* denotes the functions used in expressions, such as `CEILING()`, which calculates mathematical ceiling of a number. It does not directly reduce the number of rows, but enables pushdown when combining it with other expressions. *GroupBy* denotes the **`GROUP BY`** operators. Grouping rows in the storage layer reduced the number of rows transferred over the network. *Join* denotes **`JOIN`** operators. **`JOIN`** requires two tables, which typically exist in different nodes, making it challenging to implement pushdown. Existing methods usually implement pushdown for part of **`JOIN`**. For example, `HASH` **`JOIN`** is a two-phase algorithm that first builds a hash table on the join key of one relation and then probes it with tuples from the other relation, and PushdownDB [49] describes a method to execute the second stage in the storage layer. *Topk* denotes **`ORDER BY`** and **`LIMIT`** operators to select the top k rows, where k is a number. Enabling it would only transfer at most k rows over the network. For example, only 3 rows are transferred over the network for this query **`SELECT`** `*` **`FROM`** `employees` **`ORDER BY`** `salary` **`LIMIT`** `3`. *Union* denotes **`UNION`** operators to merge two data. Merging data before transferring over the network reduces the number of network communications.

Results. We identified 8 pushdown operators, with CockroachDB, TiDB, and YugabyteDB supporting 8, 8, and 4 of them, respectively. Overall, these DBMSs support pushdown for the majority of operators. YugabyteDB supports fewer operators, but is actively implementing other operators, such as *GroupBy*. However, because computation pushdown is implemented empirically, an operator

is typically implemented differently across DBMSs. For example, TiDB only supports a subset of functions,[1] which are different from the functions supported by CockroachDB and YugabyteDB.

Line of code. To further understand the scope of computation pushdown support, we analyzed the lines of code (LoC) associated with the computation pushdown implementations. Specifically, we measured the total LoC of source files whose names or paths contain the keywords "pushdown," "distsql," or "coprocessor," which are commonly used to denote computation pushdown in these systems. CockroachDB contains 12,524 LoC, while TiDB has 7,695 LoC. YugabyteDB does not have specific source-code files, making it difficult to estimate its pushdown-related LoC. When compared to the total codebase of each DBMS—which consists of millions of lines of code—pushdown logic constitutes only a small fraction, highlighting that it supports only a subset of SQL operators with different implementations.

Computation pushdown supports a core set of SQL operators in all studied DBMSs, but the extent and implementation details vary significantly across DBMSs.

RQ2 Scheduling Logic

We found that all 20 supported pushdown operators are scheduled by at least one scheduling function that determines whether the operator can be pushed down. Although evaluating different conditions, these scheduling functions share the same code pattern that returns a boolean value or modifies a global boolean value to indicate the eligibility of an operator for pushdown. CockroachDB implements a unified scheduling function that examines all operators in a given SQL statement and determines their feasibility for pushdown. In contrast, TiDB and YugabyteDB implement fine-grained checking mechanisms, where each pushdown operator may be governed by multiple scheduling functions. For instance, in TiDB, both *CheckAggPushDown* and *CanPushToCop* functions influence the behavior of the Aggregate pushdown operator, with the former being invoked by the latter. Although configuration options could, in principle, offer control over pushdown behavior, they work in a black-box manner, are often incomplete, and require additional developer effort. For example, in our evaluation at Table 5, 80.4% of functions are not configurable. As such, they are insufficient for systematic testing. Therefore, we propose a white-box approach: directly modifying the source code to control scheduling functions. This allows us to reliably control the behavior of all 20 operators, as confirmed by their absence in the resulting query plans.

All supported pushdown operators are governed by scheduling functions.

4 Approach

Based on our study, we propose *Controlled Pushdown Execution* (CPE), a novel white-box method to identify bugs in the computation pushdown. The core idea of CPE is to control a specific operator by modifying scheduling functions and then checking whether the system produces consistent results.

System overview. Figure 2 provides an overview of CPE, illustrated using the running example in Listing 1. First, in step ①, we use a combination of heuristic searching and validation to identify the scheduling functions associated with pushdown operators. For each identified function, we modify the source code to enforce a specific return value, so that the operator is disabled, as shown in step ③. Next, in step ②, we randomly generate a test case consisting of SQL statements to create

[1] https://docs.pingcap.com/tidb/stable/expressions-pushed-down/

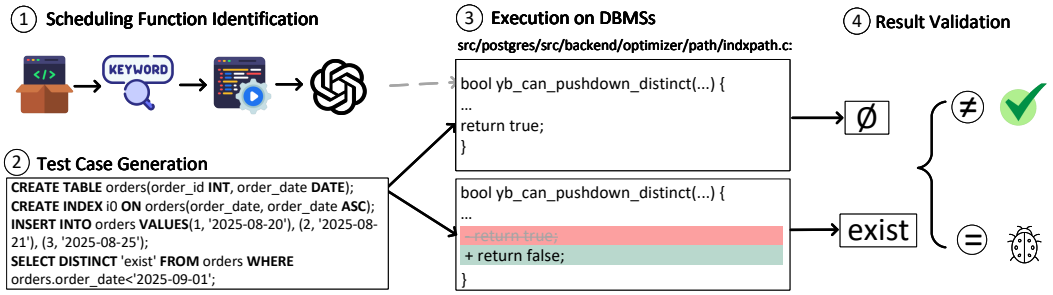


Fig. 2. Overview of CPE.

a database, insert data, and perform queries. We then execute this test case on both the original and the modified version of the DBMS in step ③. Finally, in step ④ we compare their results. A bug is found if both results are inconsistent.

4.1 Scheduling Function Identification

To identify scheduling functions in ①, we begin by performing static analysis to identify functions that are likely responsible for scheduling the execution of pushdown operators. For each candidate function, we use the official test suite to validate its functional correctness. Last, we leverage LLMs to filter out the functions for scheduling computation pushdown, instead of other behaviors. Only the functions that pass the above three checks are used in the subsequent testing phase.

Static matching. We statically identify candidate scheduling functions in the DBMS source code based on syntactic and semantic patterns we obtained from the study at Section 3. A function is considered a candidate scheduling function if it satisfies the following criteria: (1) it returns a boolean value or exclusively modifies global boolean variables, and (2) its name contains keywords indicative of pushdown-related logic, such as **`push`** or **`dist`**. These heuristics are grounded in common naming and usage conventions in DBMS implementations, where such functions are typically used to determine whether an operator (*e.g.*, expression or aggregate pushdown) can be applied. In Figure 2, we applied the query pattern `bool *push*(...)` to match the functions whose return type is boolean and whose names contain the substring "push". This static analysis provides an approximation of potential scheduling functions.

Functional validation. To avoid potential false alarms from the static matching, we perform a functional validation using the DBMS's official test suite. A function is considered a valid scheduling function if, after modifying the function to always return **`false`**, the system still passes all tests in the official suite. The rationale is that a scheduling function should only affect the presence or absence of computation pushdown, not the correctness of query results or system behavior. Modifying its decision does not break the system functionalities. We should not modify the functions to always return **`true`**, which enforces unsupported computation pushdown operators, incurring false alarms. Official test suites are typically comprehensive and well-designed, and passing all tests provides reasonable confidence that the modified function does not impact functionalities. We extract all valid scheduling functions. As shown in Figure 2, the function `yb_can_pushdown_distinct()` satisfies these criteria: it matches the static pattern, and forcing it to return **`false`** does not cause any test failures. We therefore conclude that it is a valid scheduling function.

Semantic checking. For each valid scheduling function, we further apply the LLM to identify those functions specifically responsible for scheduling computation pushdown. LLMs are powerful

tools capable of understanding both natural language and program code, which makes them well-suited for this semantic analysis task. Given the large codebase of modern DBMSs, we employ Claude Code [40] as the LLM-powered analysis tool, as it is specifically designed for autonomous repository navigation and code reasoning, and supports large context windows necessary to analyze functions across a sizable codebase. Claude Code autonomously plans how to search and navigate the repository, invokes the underlying LLM (Sonnet 4.6) with the relevant source files as context, and determines the exact steps for querying the model. After static matching and functional validation, identified functions are exported into a CSV file placed in the project root directory. Claude Code is then launched in the same directory with default settings. Our semantic-checking prompt is as follows: "*@file.csv contains a set of function names and their locations in the code repository. Please analyze these functions in the repository and indicate, for each one, whether it determines the execution of computation pushdown.*" Claude Code executes this prompt and returns the validated functions that are responsible for scheduling computation pushdown.

This approach ensures that, even if some of the identified functions are not actual scheduling functions, the program's functionality remains intact according to the official test suites. As a result, the subsequent testing remains sound, with no false positives and only one false negative observed in the evaluation in Section 5.

4.2 Testing Loop

Algorithm 1 CPE

Input: DBMS: D, scheduling functions: F

```
1:  while TRUE do
2:    Q = generate(D) // -> step ②
3:    result = execute(D, Q) // -> step ③
4:    for f ∈ F do
5:      D' = disable(D, f)
6:      result' = execute(D', Q) // -> step ③
7:      if result <> result' then
8:        save_bug_info(D, D', Q) // -> step ④
9:        exit(1)
10:     end if
11:   end for
12: end while
```

Given the scheduling functions obtained in step ①, Algorithm 1 shows the testing loop of CPE, including the steps ②, ③, and ④.

First, we pass the target DBMS D to the generator and obtain a random test case Q in line 2. A test case typically includes the SQL statements to create tables, insert data, and query data. For example, in Figure 2, the test case includes four SQL statements to create a table `t0`, insert one row with an index, and query the data from `t0`. Such test cases can be manually provided from benchmarks [22, 34] or generated [2, 15, 26, 27]. The generation of test cases is not a contribution of this paper. Unlike differential testing [4, 32] and metamorphic testing [13, 25, 26], both of which work on specific test cases, in principle, CPE can be paired with any test cases.

Then, we execute Q on D and D' to obtain the outputs *result* and *result*$'$ in lines 3 and 6. To derive each D', we disable a single scheduling function by modifying its return value or relevant global variable to always yield `false` (line 5). This corresponds to step ③ in Figure 2. For example, in the figure, the test case evaluates whether the operator `DISTINCT 'exist'` should be pushed down

to the storage engine. When executed on D, the function may return either `true` or `false`, depending on the internal scheduling logic. In contrast, when executed on D', the function deterministically returns `false`, disabling the pushdown decision. The result from this second execution serves as a reference for validating the behavior of D, which may include pushdown operators.

Lastly, we check whether $D(t) = D'(t)$ in line 7. If both are not equal, a potential bug is found. The relevant bug information is then saved for reporting (line 8), and the testing loop terminates. In Figure 2, it corresponds to the step ④. *result* and *result'* are different, so a bug is found. To facilitate bug analysis, we further minimize the test case that triggers the discrepancy, determine which execution produces the incorrect result, and report the issue to the developers.

4.3 Implementation

We implemented CPE into an automatic testing tool on top of EET [15], which includes a test case generator that can generate grammatically and syntactically correct input queries.

Static analysis for scheduling function identification. To implement the identification of scheduling functions, we utilized the static analysis tool CodeQL[2], which enables querying source code based on structural and semantic properties. We wrote CodeQL scripts to statically match candidate scheduling functions.

Eliminating recompilation. Each D' requires a recompilation of the D, and it significantly blocks the testing efficiency. To address this, we merged all D' variants into a single D' that incorporates all modifications on scheduling functions, thereby eliminating the need for recompilation. We used shared memory to achieve this. Specifically, for each scheduling function, we added a `IF` branch to return `false` or to modify the global boolean variable `false`, and the `IF` branch is controlled by the value in the shared memory. When assigning a specific value to the shared memory from CPE, the corresponding function is modified, which is equivalent to D'. In this way, we eliminate the recompilation for all D'. We implemented a Python script in around 300 lines to automatically instrument all scheduling functions in the source code.

Test case minimization. To facilitate bug analysis, we minimized the test cases using C-Reduce [24]. It follows predefined rules to remove some parts from the file while maintaining the specific property—the inconsistency of *result* and *result'*. Although originally designed for reducing C programs, C-Reduce can be effectively adapted to minimize SQL queries as well.

5 Evaluation

To evaluate the effectiveness of CPE in finding bugs in computation pushdown, we sought to answer the following questions:

Q1 Effectiveness. Can CPE find previously unknown bugs in computation pushdown?
Q2 Significance. Whether found bugs are significant compared to the historical bugs?
Q3 Necessity. Can other methods EET and TLP find these bugs found by CPE?
Q4 Soundness. Whether CPE is sound to identify bugs?

Tested DBMSs. We tested CockroachDB, TiDB, and YugabyteDB of the same versions in our study at Section 3. For CockroachDB, we deployed three nodes, each of which includes both SQL and storage layers. We configured the cluster by executing `SET distsql=always`, which enables computation pushdown for every possible SQL operator. TiDB supports two storage engines: TiKV, which is a key-value storage engine, and TiFlash, which is a column storage engine. To test both storage engines, we deployed seven nodes: one for the SQL layer, three for TiKV storage, and the other three

[2] https://codeql.github.com/

for TiFlash storage. For YugabyteDB, we deployed three nodes, each including both SQL and storage layers. We configured the cluster to start with two configurations: `yb_enable_optimizer_statistics=on` and `yb_enable_base_scans_cost_model=on`, which let query optimization consider computation pushdown, increasing the possibility of using computation pushdown.

Baselines. We compared CPE with EET [15] and TLP [26]. No prior testing method has specifically targeted computation pushdown, so we selected two state-of-the-art test oracles designed to find other types of logic bugs in DBMSs. Both EET and TLP focus on identifying bugs caused by incorrect expression processing, rather than computation pushdown. EET transforms an expression into a semantically equivalent one and checks whether both yield the same result. TLP generates multiple, more complex queries that each compute a partition of the original query's result; the oracle then checks whether the combined results are equivalent to the original. TLP is implemented in the framework SQLancer[3], which supports more test oracles to find logic bugs. We did not consider other oracles in SQLancer, because they do not support all three evaluated DBMSs yet. Both baselines have been widely used and have found thousands of bugs in real-world DBMSs. By comparing with them, we gain insights into the necessity of CPE against other testing methods for finding bugs in computation pushdown of distributed DBMSs.

Experimental infrastructure. We conducted all experiments on an AMD Ryzen Threadripper 3990X processor that has 64 physical and 128 logical cores clocked at 2.2 GHz. Our test machine uses Ubuntu 24.04 with 256 GB of RAM. We limit our maximum utilization to 20 cores to avoid resource competition.

Q1 Effectiveness

We ran CPE during approximately two months—during which we also implemented the approach—aiming to find bugs in the computation pushdown. To avoid burdening developers, we only report the bugs that we suspect to be unique. Developers typically directly reply whether it is a unique bug.

Finding overview. Table 2 lists the unique and previously unknown bugs found by CPE. The *Bug ID* column lists the identifiers in each DBMS's bug tracker. In the *Category* column, *Logic* denotes logic bugs, whereas *Crash* covers crash or error bugs, which cause the system to abort or emit an error message. For logic bugs, the *Operator* column specifies the associated computation pushdown operator. In total, CPE found 25 bugs, all of which had been confirmed by developers with no false alarm, and 9 bugs had been fixed. Of the 14 logic bugs, 13 are due to incorrect pushdown implementation. The remaining logic bug is caused by an erroneous hash-join algorithm, as shown in Listing 6; we discuss this case in Section 6.

Severity. Within 25 found unique bugs, 9 bugs (6 logic bugs and 3 crash bugs) were assigned *Major* or *High* severity, indicating these bugs are important and emergency. 10 bugs were assigned *Moderate* or *Medium*, indicating these bugs are important but not emergency. Only one bug was assigned *Minor*, which refers to a corner case. The other bugs were not assigned a severity level.

Bug diversity. In total, 13 logic bugs involve 5 out of the 8 pushdown operators that we studied in Table 1, demonstrating that CPE is effective at uncovering logic bugs across a range of operators. The *Expression* and *Function* operators revealed the most bugs, likely because they support a broader and more flexible set of input queries compared to the others.

[3] www.sqlancer.com/

Table 2. Previously unknown and unique bugs found by CPE.

DBMS	Bug ID	Category	Operator	Status
CockroachDB	145108	Logic	Expression	Confirmed
CockroachDB	148279	Logic	Expression	Confirmed
CockroachDB	145888	Crash	-	Fixed
CockroachDB	146065	Crash	-	Fixed
TiDB	60586	Logic	Expression	Fixed
TiDB	60624	Logic	Function	Confirmed
TiDB	60630	Logic	Expression	Confirmed
TiDB	60664	Logic	Function	Confirmed
TiDB	60674	Logic	-	Confirmed
TiDB	60841	Logic	Function	Confirmed
TiDB	60958	Logic	Join	Confirmed
TiDB	60960	Logic	Function	Confirmed
TiDB	61664	Logic	Aggregate	Confirmed
TiDB	61683	Crash	-	Confirmed
TiDB	61684	Crash	-	Fixed
TiDB	61694	Crash	-	Confirmed
TiDB	61715	Crash	-	Fixed
TiDB	61735	Crash	-	Fixed
YugabyteDB	26620	Logic	Aggregate	Confirmed
YugabyteDB	26717	Logic	Distinct	Fixed
YugabyteDB	27031	Logic	Expression	Fixed
YugabyteDB	26753	Crash	-	Confirmed
YugabyteDB	27009	Crash	-	Fixed
YugabyteDB	27029	Crash	-	Confirmed
YugabyteDB	27059	Crash	-	Confirmed
			Sum:	25

Listing 2. A bug in the function operator in TiDB.

```
1 CREATE TABLE t0(c0 CHAR(1));
2 INSERT t0 VALUES('1');
3 ALTER TABLE t0 SET TIFLASH REPLICA 1;
4 SELECT * FROM t0 WHERE RIGHT(c0, -1 | 0) IS NOT NULL; -- {} ✔ without pushdown
      {1} 🐞 with pushdown
```

Listing 3. A bug in the expression operator in CockroachDB.

```
1 CREATE TABLE t0(c0 TEXT);
2 INSERT INTO t0 VALUES('/a');
3 SELECT * FROM t0 JOIN (SELECT 1) ON NOT(-61.100 // 79 || c0 < c0);  -- {} ✔
      without pushdown {/a|1} 🐞 with pushdown
```

Example 1: a bug in the function operator for pushdown. Listing 2 shows a logic bug that CPE found in TiDB. Lines 1 and 2 create a table and insert a single row. Line 3 enables the TiFlash storage engine for the table via an `ALTER` statement. The query in line 4 uses the `RIGHT()` function, which extracts the rightmost "`-1 | 0`" characters of `c0`. Since `-1 | 0` evaluates to `-1`, the expected behavior is for `RIGHT()` to return `NULL`, causing the query to return no rows. However, due to a buggy implementation of `RIGHT()` in the storage engine, it incorrectly returns a non-`NULL` value, and the query yields one row unexpectedly. CPE identified the issue by modifying the scheduling function `canFuncBePushed()` to return `false`, forcing the query to bypass the pushdown logic. As a result, the query returns an empty result set, thereby exposing the bug.

Table 3. Historical bugs in computation pushdown.

DBMS	Bug ID	CPE
CockroachDB	144384	✓
TiDB	48501	✓
TiDB	44135	✓
TiDB	45550	✓

Example 2: a bug in the expression operator for pushdown. Listing 3 presents another bug that CPE found in CockroachDB. Lines 1 and 2 create a table and insert a single row. The query in line 3 performs a `JOIN` between `t0` and `SELECT 1`, with the join condition `NOT(-61.100 // 79 || c0 < c0)` being eligible for pushdown to the storage engine. The `//` operator denotes integer division that rounds the result to the nearest integer. `-61.100 // 79` evaluates to `-0.0`. According to the IEEE 754 standard [1], `0.0` and `-0.0` are distinct values, so `-0.0` is a non-zero value and is evaluated `TRUE`, the overall expression in the `WHERE` clause should evaluate to `FALSE`, and the query is expected to return an empty result. However, due to an incorrect conversion of `-0.0` to `0.0` in the storage layer, the expression is mis-evaluated, and the query unexpectedly returns one row. CPE exposed this bug by modifying the scheduling function `checkSupportForPlanNode()` to always return `false`, thereby disabling pushdown. As a result, the query produces the correct (empty) result, exposing this bug.

CPE found 25 unique and previously unknown bugs, in which 14 are logic bugs, covering 5 of 8 pushdown operators.

Q2 Significance

To understand the significance of the bugs found by CPE, we studied the historical bugs in computation pushdown, and evaluated whether CPE can reproduce them. We searched the GitHub issues of each DBMS, in which users typically report bugs. Specifically, we searched the keywords "pushdown" and "distsql", both of which represent the computation pushdown in studied DBMSs. From the matched issues, we manually filtered out logic bugs specific to computation pushdown. For every candidate bug, we re-ran its published test case to confirm that the problem was unique and still observable (*e.g.*, YugabyteDB issue #24144 could not be observed because its benchmark is no longer available). Finally, we instrumented the relevant scheduling functions with CPE, disabled each in turn, and observed whether the same discrepancy resurfaced. If it did, we deemed the historical bug reproducible by CPE.

Results. Table 3 shows the historical logic bugs related to computation pushdown. In total, only four such bugs have been identified, highlighting the lack of effective testing approaches for pushdown implementations. CPE is the first approach specifically designed to validate computation pushdown, uncovering significantly more bugs—three times as many as previously known—and successfully reproducing all historical bugs.

Example. Listing 4 shows the example of the bug #48501 in TiDB. The root reason is an incorrect implementation of the function `date()` for computation pushdown. The query in line 3 should return one row. However, when pushing the function `date()` down to the storage layer, the query returns an empty result. CPE modified the scheduling function `canExprPushDown()` to disable the pushdown of `date()`, and this query returns one row as expected. In TiDB, the storage layer is implemented in Rust, while the SQL layer is implemented in Go. This language mismatch adds complexity to the correct implementation of computation pushdown.

Listing 4. Historical bug 48501 in pushdown.

```
1  CREATE TABLE t(a INT, b YEAR);
2  INSERT INTO t VALUES (1, 2021);
3  SELECT * FROM t WHERE date(b); --  {1|2010} ✔ {} 🐞
```

CPE found 3 × more bugs than historical logic bugs in computation pushdown, and can reproduce all of them.

Q3 Necessity

To assess the necessity of CPE in uncovering the newly found logic bugs listed in Table 2, we compared it with EET and TLP through the following three experiments.

(1) We evaluated whether EET and TLP can find the bugs found by CPE. The necessity is high if the bugs found by CPE are seldom found by EET and TLP. Since CPE executes a single test case across different program variants to check behavioral consistency, while EET and TLP execute pairs of semantically equivalent queries on the same program, a direct empirical comparison without transformation is fundamentally infeasible. Following the methodology of prior work [4, 25, 26], we manually translated each CPE test case into equivalent EET and TLP test cases and checked whether those variants exposed the same discrepancy. This transformation establishes an *upper bound* on whether EET or TLP could detect the same bugs under ideal conditions; in practice, their effectiveness would likely be lower due to the randomness inherent in test case generation. For TLP, if a test case includes an expression, we split it into three parts as described by TLP [26]. For EET, the available equivalent expression space is too huge to enumerate, so we asked ChatGPT [39] to randomly generate a translation. While we cannot completely rule out misclassifications that might be due to overlooking that a bug could be found by another method, we believe that the majority of cases were clear.

Fairness limitations. We acknowledge that our comparison reflects the *typical* behavior of EET and TLP rather than their *theoretical maximum*, and results should be interpreted accordingly. For EET, the translated equivalent expression is a single random sample from a vast space, so a more exhaustive or targeted search might occasionally find a translating expression that does expose a discrepancy; for TLP, our translation follows the standard three-way partition, but alternative partitioning strategies are possible.

Results. Among the 14 logic bugs found by CPE, none can be identified by EET, and 12 cannot be identified by TLP. For EET, every translated pair of queries produced identical results, because both equivalent expressions were executed in the same location of SQL or storage layers. For TLP, five test cases could not be translated into TLP test cases due to the lack of expressions. The remaining 7 test cases showed no discrepancy due to a similar reason as EET. Only two bugs were detectable by TLP, because the paired queries happened to be executed in different locations (SQL and storage layers). However, TLP does not explicitly control or enforce the execution location, making such detections accidental rather than systematic. Hence, CPE is essential for finding most of these bugs.

Example. Listing 5 shows an example of translating CPE test case in Listing 1 to equivalent test cases for EET and TLP. The first three statements can be directly applied to both methods. For EET, the first query is the same as the original query in line 4 of Listing 1, and the second query is a semantic-equivalent derivation of the first query by changing the expression E to *false OR* E, in which *false* refers an expression that is always evaluated to false. Both E and *false OR* E are executed

Listing 5. Equivalent test case of Listing 1 for EET and TLP.

```
CREATE TABLE t0(c0 INT);
CREATE INDEX i0 ON t0(c0, c0 ASC);
INSERT INTO t0(c0) VALUES(1);
-------------------------------------EET-------------------------------------
SELECT DISTINCT 'a' FROM t0 WHERE t0.c0<5; -- {}
SELECT DISTINCT 'a' FROM t0 WHERE ((t0.c0 = 100) AND NOT (t0.c0 = 100) AND
    (t0.c0 = 100 IS NOT NULL)) OR (t0.c0 < 5); -- {}
-------------------------------------TLP-------------------------------------
SELECT DISTINCT 'a' FROM t0; -- {}
SELECT DISTINCT 'a' FROM t0 WHERE t0.c0<5 IS TRUE UNION ALL SELECT DISTINCT
    'a' FROM t0 WHERE t0.c0<5 IS FALSE UNION ALL SELECT DISTINCT 'a' FROM t0
    WHERE t0.c0<5 IS NULL; -- {}
```

in the storage layer, so both queries return the same incorrect results, so we cannot observe a discrepancy. For TLP, the first query is generated by omitting the **WHERE** in the original query, and the second query is a union of three queries with different expressions in **WHERE**. Similarly, both queries are executed in the storage layer, and we cannot observe a discrepancy. Without modifying the function `yb_can_pushodwn_distinct()` to disable the computation pushdown of the operator **DISTINCT**, EET and TLP failed to find this bug. Neither EET nor TLP can detect this bug, because they lack the mechanism to force execution across different layers that CPE exploits.

> Out of the 14 logic bugs found by CPE, none can be identified by EET, and only 2 can be identified by TLP.

(2) We evaluated whether CPE find more unique bugs in computation pushdown than EET and TLP in 24 hours. The necessity is high if so. To do this, we ran CPE, EET, and TLP for 24 hours and 10 runs. We used a best-effort approach to identify unique logic bugs. For EET and TLP, we compared the SQL clause structures—treating bugs as duplicates if their queries shared the same combination of clauses (*e.g.*, both involving only **RIGHT JOIN** and **GROUP BY**). For CPE, we deemed the bug-inducing test cases that show a discrepancy when changing the same scheduling function as duplicates. Then, for each unique bug, we examined whether it is in the computation pushdown. For CPE, a bug is related to pushdown if modifying any scheduling function affects its buggy behavior. For EET and TLP, it is challenging to examine this due to a lack of expert knowledge. To solve this, similar to CPE, we examined whether the bug behavior is affected when modifying any scheduling function. Although this classification involves manual judgment, we believe that the vast majority of cases are labeled correctly.

Results. Figure 3 shows the average number of unique logic bugs found by CPE, EET, and TLP over 24 hours across 10 runs. *All* shows the total number of unique logic bugs found, while *Pushdown* indicates the subset related specifically to computation pushdown. Each bar indicates the average number of unique bugs discovered per 24-hour test cycle across ten independent runs, with the error bars denoting the full min-max range. Overall, CPE discovered 5× more unique logic bugs than EET and TLP in computation pushdown. In contrast, EET and TLP found more logic bugs outside of computation pushdown, which is expected since neither method is designed to test computation pushdown.

Code coverage. To further assess bug-finding efficiency, we examined code coverage of CPE, EET, and TLP in computation pushdown. We only evaluated TiDB because CockroachDB does not support

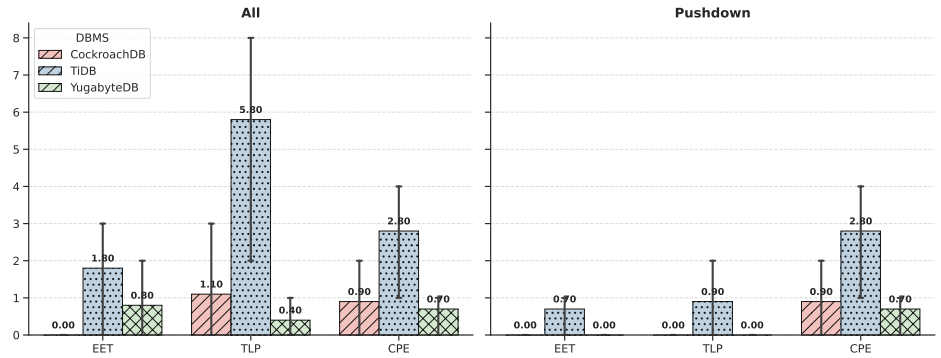


Fig. 3. Average number of unique bugs found in 24 hours across 10 runs.

Table 4. The number of bugs existed before the publication of EET and TLP.

DBMS	Bugs	Before TLP	Before EET
CockroachDB	2	0	2
TiDB	9	4	8

code coverage measurement for end-to-end tests.[4], and it is unclear which code in YugabyteDB is responsible for the computation pushdown. We acknowledge that drawing conclusions from a single DBMS is a limitation; however, TiDB represents the only system among the three for which coverage measurement is both technically feasible and scoped to a well-defined pushdown module, and we therefore treat these results as indicative rather than definitive. Due to the limitations of the code coverage tool in Go language, we can only measure the statement coverage for each package. Therefore, we measured the statement coverage in the package *"pkg/distsql"*, which includes the most code for the computation pushdown. The total LoC is 3,805. On average, EET, TLP, and CPE cover 59.85%, 60.42%, and 63.94% statement coverage in computation pushdown in 24 hours across 10 runs. While CPE is not designed to maximize coverage, it achieves slightly higher coverage than the baselines. This is expected as CPE disables computation pushdown operators to generate reference outputs, exercising more code.

CPE finds 5× more unique logic bugs than EET and TLP in computation pushdown.

(3) We evaluated whether the bugs found by CPE have existed before EET and TLP. The necessity of CPE is high if it finds bugs that have existed despite the extensive testing efforts of EET and TLP. To evaluate this, we examined whether a bug existed in a DBMS version released before the publication dates of EET (July 2024) and TLP (November 2020). If so, we consider the bug to have been missed by those methods, and CPE is necessary to find these bugs. We excluded YugabyteDB, as it was not evaluated by either EET or TLP, while both CockroachDB and TiDB have been extensively tested by them.

Results. Table 4 shows the result. Within 11 unique logic bugs found by CPE in CockroachDB and TiDB, 4 existed before the publication of both TLP and EET, and 6 existed before that of EET only. The result indicates that most of these bugs had been present for a long time and were not uncovered despite extensive testing by TLP and EET. The lower bug number of bugs existed

[4]https://cockroachlabs.atlassian.net/wiki/spaces/CRDB/pages/73171260/Code+coverage

Table 5. Identified scheduling functions by CPE.

DBMS	All	Pushdown	Configurable
CockroachDB	12	12	2
TiDB	26	25	5
YugabyteDB	13	13	3
Sum:	51	50	10

before the publication date of TLP is expected, as TLP was published nearly five years ago, when computation pushdown was still in its early stages of development.

10 of 11 newly found logic bugs in CockroachDB and TiDB have existed before the publication date of TLP or EET.

Q4 Soundness

Function identification accuracy. We evaluated the accuracy of the scheduling function identification by analyzing potential false positives and false negatives. To measure false positives, the non-scheduling functions identified by CPE, we manually examined the source code, with particular focus on comments. Measuring false negatives, the scheduling functions that CPE missed, is more challenging due to the lack of expert knowledge on the source code. To solve this, we examined whether the functions identified by CPE fail to control any of the supported operators in Table 1. Additionally, to evaluate whether code modification is necessary, we examined whether the identified functions can be controlled by external configurations.

Results. Table 5 shows the number of functions identified by CPE. *All* column refers to the number of all identified functions, *Pushdown* refers to the number of identified functions related to pushdown, and *Configurable* refers to the number of identified functions that can be controlled by external configurations. Among the 51 identified functions, 50 (98%) are true scheduling functions for computation pushdown. The only false positive is a scheduling function for switching between different versions of the hash join algorithm. It involves join expression pushdown, a SQL optimization with a name similar to computation pushdown, which may cause LLMs to misclassify it as a scheduling function for computation pushdown. Such functions influence system behavior in ways that are difficult to control through inputs alone, similar to pushdown. Notably, instrumenting these functions did not break functionalities, as all changes passed the official test suites. We further discuss this in Section 6. Regarding false negatives, all supported operators can be controlled by the identified functions, resulting in a rate of zero. Finally, only 10 functions (19.6%) are directly controllable via external configuration, highlighting the necessity to control pushdown by code modification.

Effectiveness of the identification pipeline. To assess the contribution of each step in pushdown function identification, we compared the number of candidate functions obtained after static matching, functional validation, and semantic checking, as well as the effectiveness of directly applying LLMs to replace all three steps. The identification pipeline consists of three stages:

(1) *Static matching* selects the functions whose names or signatures suggest scheduling relevance. It yields 59, 114, and 62 candidate functions in CockroachDB, TiDB, and YugabyteDB, respectively.
(2) *Functional validation* eliminates syntactically plausible but semantically irrelevant candidates by checking behavioral properties via dynamic analysis. It reduces the candidates to 34, 30, and 26 functions, eliminating approximately 62% of false candidates.

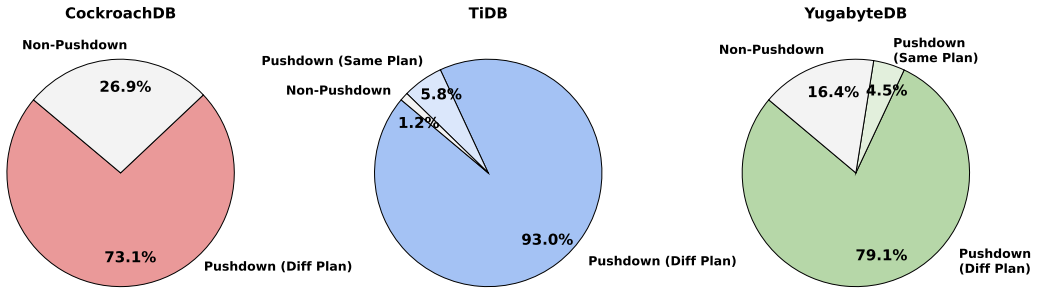


Fig. 4. Percentage of executions that trigger pushdown and expose differences under CPE.

(3) *Semantic checking* applies LLM-based reasoning to make a final, fine-grained determination of whether each remaining function governs computation pushdown. It further filters out about 43% of the remaining false functions, yielding 12, 26, and 13 confirmed functions.

In contrast, directly prompting Claude Code with "*Please analyze this code repository to identify all scheduling functions that determine the execution of computation pushdown.*" achieves markedly lower accuracy: it correctly identifies only 8, 4, and 14 pushdown functions in the three DBMSs, while introducing 5, 14, and 1 false positives for the three DBMSs, respectively. These results highlight the necessity of a multi-step identification pipeline, as relying on LLMs alone remains insufficient for accurate pushdown function detection.

Stability of semantic checking across LLMs. To evaluate the stability of the semantic checking step with respect to the choice of LLMs, we evaluated two more LLM agents: Gemini CLI (Gemini 3.1 Pro) and Copilot (GPT 5.4). For CockroachDB, TiDB, and YugabyteDB, Gemini CLI identified 12, 26, and 13 scheduling functions, respectively, matching the results reported by Claude Code. Copilot identified 11, 26, and 14 functions, missing one true pushdown function in CockroachDB and introducing one false positive in YugabyteDB. As modifications to these functions pass functional validation, this false positive does not disrupt the pipeline during testing. These results suggest that the semantic checking step is largely stable across state-of-the-art LLM agents, with minor variance attributable to differences in code reasoning capabilities.

Pushdown coverage and oracle non-triviality. We evaluated to what extent computation pushdown is triggered and whether CPE produces non-trivial differential checks when it is. A higher percentage indicates greater effectiveness of CPE in validating the consistency of enabling and disabling computation pushdown. To determine whether computation pushdown is triggered, we examined the query plans of all queries across 10 24-hour runs, using the same methodology as in Section 3. For each plan, we further checked whether disabling the relevant scheduling function produced a different query plan. Figure 4 reports both results. Overall, 73.1%, 98.8%, and 83.6% of executed queries triggered computation pushdown in CockroachDB, TiDB, and YugabyteDB, respectively. Among these, CPE performs a non-trivial differential check on 73.1%, 93.0%, and 79.1% of all executed inputs across the three DBMSs. Restricting to queries that trigger pushdown, 100%, 94.4%, and 94.6% queries produced a different query plan when CPE disabled a scheduling function. It shows that CPE is efficient to validate computation pushdown. At most 6% of queries that trigger pushdown produce unchanged plans after disabling a scheduling function, indicating that only a small minority of pushdown decisions are not exclusively governed by such functions.

Listing 6. A found bug not related computation pushdown.

```
CREATE TABLE t1 (c0 int, c1 int);
INSERT INTO t1 VALUES (1, 0), (1, 0);
CREATE TABLE t2 (c0 int, c1 int);
INSERT INTO t2 VALUES (1, 0);
SELECT c0 FROM t2 ref_0 WHERE ref_0.c1 IN (SELECT t1.c1 FROM t2 JOIN t1 WHERE
    ref_0.c1 OR TRUE) --  {0} {0} ✔ {0} 🐞
```

6 Discussion

From testing to fix. Each code modification introduced by CPE can serve as a coarse-grained program fix. After we reported a bug, it typically took several weeks for developers to analyze and deploy a proper fix. During this period, however, the system remained fragile. Existing testing methods can only reveal such bugs but offer no mitigation. In contrast, CPE detects bugs by modifying scheduling functions related to computation pushdown, enabling a way to temporarily avoid triggering the bug. Specifically, we can directly apply our modification to forcibly disable the pushdown of a problematic operator. While coarse-grained, this fix can serve as a practical interim solution. Developers sometimes adopt similar strategies. For instance, in CockroachDB, the bug #146898 was caused by an incorrect implementation of pushdown for the function `to_regclass()` to the storage layer. To resolve it, developers concluded that this function could not be safely pushed down and disabled its pushdown.

Path to adoption. We envision CPE becoming widely adopted beyond computation pushdown. As noted in Section 5, some scheduling functions CPE govern other system behaviors, allowing CPE to reveal bugs in other system components. For example, Listing 6 shows a TiDB bug. The failure stems from an incorrect implementation of TiDB's hash-join algorithm. TiDB implements two hash-join executors: v1 and v2. It uses the scheduling function `canUseHashJoinV2()` to decide between them: if its preconditions hold, v2 is chosen; otherwise, the engine falls back to v1. By modifying this function, CPE found this bug. While CPE targets computation pushdown in this work, its core insight—making execution decisions explicitly controllable—applies more broadly. For instance, transaction scheduling, memory layout optimizations, or query plan selection often involve heuristic or opaque decisions within DBMSs. By instrumenting these decision points with expert knowledge, we can create new oracles for revealing subtle logic bugs that evade traditional input-based testing. We believe this direction holds promise for systematically testing complex, stateful systems with implicit behaviors.

Generalizability and Assumptions. While CPE demonstrates effectiveness across the three evaluated DBMSs, several assumptions underlying the methodology warrant discussion. First, our approach requires access to the full source code of the target DBMS, which excludes closed-source or proprietary systems. Second, the syntactic heuristic used to identify candidate scheduling functions relies on naming conventions (*e.g.*, function names containing terms such as `push` or `dist`) that are common in well-maintained, open-source codebases but may not generalize to systems with inconsistent or opaque naming practices. Third, our method assumes that computation pushdown scheduling logic is encapsulated in discrete, identifiable functions rather than being scattered across deeply nested or macro-generated code, which reflects the internal structure of the three systems studied but may not hold universally. Fourth, the quality and coverage of the official test suite directly affect the fidelity of our dynamic analysis phase: a sparse or insufficiently covering test suite may fail to exercise relevant scheduling paths, leading to missed candidates. For our evaluated DBMSs: CockroachDB, TiDB, and YugabyteDB, the line coverage of their official test

suites is 57.28%, 64.39%, and 75.41%, respectively, indicating a reasonable degree of confidence in the dynamic analysis results. Finally, we acknowledge that the syntactic heuristic was both derived and evaluated on the same three DBMSs, which introduces a risk of overfitting to the conventions of these particular systems. Future work should validate the approach on additional DBMSs.

Limitations. Despite its effectiveness, CPE has several limitations. First, the identification of scheduling functions relies on heuristics, which may not be entirely complete or precise. To mitigate this, we employed functional validation using official test suites and semantic checking using LLMs to filter out false positives. Our analysis confirmed that all supported pushdown operators can be controlled through the identified functions. Moreover, this limitation can likely be addressed with lightweight expert effort. For example, our manual analysis for confirming the scheduling functions in Table 5 was completed by a single author in one day. Second, CPE detects bugs by comparing executions with and without pushdown, and thus cannot reveal bugs that manifest identically in both execution paths.

7 Related Work

Computation pushdown. The research on computation pushdown has been conducted since the 1970s. The early DBMSs pushed computation to the storage layer via special hardware. IDM [33] pushes simple operations (*e.g.*, filters) close to the disks. Grace [10] adopts parallel processors, each of which is connected to a disk module to improve the efficiency of computation pushdown. IBM Netezza data warehouse [8] designs FPGA-enabled processors for computation pushdown. Oracle [42] proposes a storage unit that natively supports computation pushdown. Nowadays, distributed DBMSs widely adopt the computation pushdown. PushdownDB [49] proposes a computation pushdown implementation for Amazon Web Service. Adaptive Pushdown [48] considers resource utilization status when scheduling the computation pushdown for a better performance. FlexPushdownDB [47] combines computation pushdown and data cache to avoid unnecessary computation pushdown. In this work, instead of optimizing computation pushdown, we studied the computation pushdown in real-world DBMSs and proposed a novel approach to find bugs in it.

White-box testing methodology. Within the broader software testing literature, CPE is a complementary white-box testing methodology. Most automated testing techniques treat the system under test as a black box, reasoning solely over externally observable inputs and outputs. Differential testing [20] and metamorphic testing [6] both follow this model: they manipulate inputs and check output consistency, without any visibility into or control over internal execution. Mutation testing [12] is the most prominent white-box exception, introducing source-level faults to probe test-suite sensitivity, but its goal is to evaluate tests rather than to find bugs. Coverage-guided fuzzing [21, 35] occupies a middle ground, using internal instrumentation as feedback to steer input generation, yet still targets the system externally through its input interface. CPE proposes a different white-box testing paradigm: rather than modifying inputs or measuring coverage, it directly transforms internal execution semantics while holding the workload fixed, and uses output consistency as the correctness oracle. This design separates the test transformation from the input space entirely, enabling bug detection that is orthogonal to and composable with existing input-generation techniques.

DBMS testing. Multiple methods have been proposed to find crash and logic bugs in DBMSs. Fuzzing is a powerful technique to randomly generate test cases for finding crash bugs. Tools such as SQLSmith [36], Griffin [9], DynSQL [14], BuzzBee [46], and ADUSA [18] employ grammar-based approaches to create test cases targeting crashes. Differential testing[20] is a widely used strategy to find logic bugs by comparing the outputs of multiple systems. This technique has been

applied effectively in various contexts, such as comparing different DBMS implementations[11, 29, 37] or different versions of the same DBMS [4, 43, 44]. While these approaches have proven successful in uncovering bugs, they often suffer from false positives due to discrepancies in SQL dialects or intentional behavioral differences between versions. Metamorphic testing[6] offers another powerful way to find logic bugs by generating semantically related test-case pairs and verifying result consistency between them. For instance, TLP [26] focuses on detecting bugs in `SELECT` statements, while DQE [30] targets logic errors in `UPDATE` and `INSERT` operations. In this work, we introduced a white-box method tailored to identifying logic bugs in the computation pushdown of distributed DBMSs.

Distributed system testing. While no prior work explicitly targets the testing of distributed DBMSs, much of the research on distributed systems applies to the underlying protocols used by such DBMSs. This body of work primarily focuses on stress testing, which involves injecting faults, such as network partitions or node crashes, into a distributed system and checking its behavior against a predefined specification. One of the most widely used frameworks for this purpose is Jepsen [41], which has proven effective in uncovering consistency violations in distributed DBMSs [19]. Elle [16], building on Jepsen, defines formal specifications to detect isolation violations. Modist [45] intercepts the network layer to inject faults, while ModP [7] introduces a testing-friendly language for implementing distributed systems. In contrast to these approaches, our work focuses on the SQL-related execution of distributed DBMSs. We propose a novel method for uncovering bugs in its core component, computation pushdown.

Intramorphic testing. At a conceptual level, CPE falls under the category of intramorphic testing [28], a methodology that generates semantically related program pairs and checks the consistency of their results. Intramorphic testing transforms a given system P into a variant P', and a test oracle is used to determine whether the outputs of P and P' satisfy a specified relation. In our approach, CPE modifies the checking functions within P to produce P', with the relation being that both versions should yield identical results. While intramorphic testing currently exists as a conceptual methodology, CPE is a concrete instantiation of this methodology, aimed at addressing a practical problem, *i.e.*, reliability of distributed DBMSs.

Mutation testing. Mutation testing [12] is a technique used to evaluate the effectiveness of a test suite. It works by intentionally introducing small, often malicious or incorrect changes into the program's source code and checking whether the existing test cases can detect these changes by causing the program to fail. While CPE also modifies the program code, its purpose is fundamentally different: the modifications are non-malicious and semantics-preserving, aimed solely at disabling specific behaviors (*e.g.*, scheduling decisions) to find bugs, rather than to assess test suite adequacy.

8 Conclusion

In this paper, we have studied the computation pushdown, a critical technique in distributed DBMSs, and presented a novel white-box methodology, *Controlled Pushdown Execution* (CPE), to validate its correctness. CPE modifies the DBMS source code to disable a specific pushdown operator and then compares the resulting query outputs; any inconsistency indicates a bug. We evaluated CPE across three widely-used distributed DBMSs: CockroachDB, TiDB, and YugabyteDB. CPE found 25 unique and previously unknown bugs, including 14 logic bugs, 13 of which are related to computation pushdown. CPE found 3× more bugs than historical bugs, and can reproduce all of them. Other methods, EET and TLP, cannot find 12 of 14 logic bugs found by CPE. Owing to its simplicity and effectiveness, we believe CPE can enhance DBMS reliability in practice and open a promising research direction—fine-grained analyzing system behaviors through source-code modification.

9 Data Availability

The artifact is available at https://zenodo.org/records/21383195.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008). https://doi.org/10.1109/IEEESTD.2019.8766229 https://standards.ieee.org/standard/754-2019.html.

[2] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174

[3] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 917–917. https://doi.org/10.1145/3597503.3639076

[4] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[7] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[8] Phil Francisco et al. 2011. *The Netezza data appliance architecture: A platform for high performance data warehousing and analytics*. IBM Redbooks.

[9] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 49:1–49:12. https://doi.org/10.1145/3551349.3560431

[10] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE.. In *VLDB*, Vol. 86. 209–219.

[11] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: searching correctness regressions in modern query engines. In *Proceedings of the workshop on Testing Database Systems*. 1–6.

[12] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[13] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 46:1–46:12. https://doi.org/10.1145/3597503.3623307

[14] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. (Aug. 2023).

[15] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835.

[16] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554* (2020).

[17] Doug Laney et al. 2001. 3D data management: Controlling data volume, velocity and variety. *META group research note* 6, 70 (2001), 1.

[18] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 225–236. https://doi.org/10.1145/3510003.3510093

[19] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–24.

[20] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[21] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[22] Raghunath Othayoth Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Matthew Emmerton, Forrest Carman, and Michael Majdalany. 2012. TPC Benchmark Roadmap 2012. In *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7755)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-642-36727-4_1

[23] Saeed K Rahimi and Frank S Haug. 2015. *Distributed database management systems: A Practical Approach*. John Wiley & Sons.

[24] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[25] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States) *(ESEC/FSE 2020)*. https://doi.org/10.1145/3368089.3409710

[26] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). https://doi.org/10.1145/3428279

[27] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta.

[28] Manuel Rigger and Zhendong Su. 2022. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 128–136.

[29] Donald R. Slutz. 1998. *Massive Stochastic Testing of SQL*. Technical Report MSR-TR-98-21. 9 pages. https://www.microsoft.com/en-us/research/publication/massive-stochastic-testing-of-sql/

[30] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*.

[31] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel "big data" science and technology center vision and execution plan. *SIGMOD Rec.* 42, 1 (2013), 44–49. https://doi.org/10.1145/2481528.2481537

[32] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26. https://doi.org/10.1145/3588909

[33] Michael Ubell. 1985. The intelligent database machine (IDM). In *Query processing in database systems*. Springer, 237–247.

[34] Website. 1988. TPC-DS Benchmark. https://www.tpc.org/tpcds/. Accessed: 2022-11-15.

[35] Website. 2013. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2025-06-20.

[36] Website. 2015. SQLsmith. https://github.com/anse1/sqlsmith. Accessed: 2025-06-20.

[37] Website. 2022. Sqllogictest Document. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. Accessed: 2025-06-20.

[38] Website. 2022. What is Database. https://www.oracle.com/database/what-is-database. Accessed: 2022-11-15.

[39] Website. 2025. ChatGPT. https://chatgpt.com/. Accessed: 2025-06-20.

[40] Website. 2025. Claude Code. https://www.anthropic.com/claude-code. Accessed: 2025-09-08.

[41] Website. 2025. Jepsen Testing Framework. https://jepsen.io/

[42] Ronald Weiss. 2012. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores* (2012).

[43] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. 2008. Oracle's SQL Performance Analyzer. *IEEE Data Eng. Bull.* 31, 1 (2008), 51–58.

[44] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[45] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI 2009*. 213–228.

[46] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. 2024. Towards generic database management system fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 901–918.

[47] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB*

Endowment 14, 11 (2021).

[48] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2023. Enhancing Computation Pushdown for Cloud OLAP Databases. *arXiv preprint arXiv:2312.15405* (2023).

[49] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.

[50] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* 1, 2 (2023).

Received 2026-01-24; accepted 2026-06-25