BranchBench: An Extensible Benchmark for Agentic Database Branching

Elaine Ang
Columbia University
ra3448@columbia.edu

Sam Weldon
Columbia University
sw3927@columbia.edu

In Keun Kim
Columbia University
ik2619@columbia.edu

Kevin Durand
Columbia University
kpd2136@columbia.edu

Kostis Kaffes
Columbia University
kkaffes@cs.columbia.edu

Eugene Wu
Columbia University
ewu@cs.columbia.edu

ABSTRACT

Agentic systems increasingly use databases as persistent working memory for speculative action: they fork state, mutate it, evaluate outcomes, and discard or retain candidate trajectories. Existing relational benchmarks assume a single shared database state and therefore do not measure the branch lifecycle, branch-tree scalability, cross-branch comparison, or resource costs induced by these workloads. We present BranchBench, a benchmark framework for agentic database branching. BranchBench makes speculative database workloads reproducible and comparable by separating workload topology, branch lifecycle operations, branch-local SQL, cross-branch comparison, and pruning into explicit parameters. Macrobenchmarks instantiate complete branch–mutate–evaluate–prune workflows, while microbenchmarks isolate the operations needed to explain end-to-end behavior. Running BranchBench on Neon and Dolt shows how the benchmark can identify backend fit for a given workload while exposing architectural trade-offs. In this case, BranchBench reveals a branch-agility versus branch-local-query trade-off: Neon favors fewer branches with data-intensive queries, while Dolt favors many branches with lightweight data operations.

CCS CONCEPTS

• **Information systems → Database performance evaluation**; *Database management system engines.*

KEYWORDS

database branching, benchmark, agentic systems

1 INTRODUCTION

Large language model agents are changing database workloads from operations over one evolving database state into *agentic speculation*: constructing and evaluating many hypothetical states. These workloads combine branch creation, connection isolated schema and data mutations, branch-local evaluation, cross-branch comparison, and pruning over wide or deep exploration trees. Monte Carlo Tree Search (MCTS), for example, selects, expands, and scores branches over database-backed state, improving Terminal-Bench success rates from 3.4% to 30.6% by exploring alternative action trajectories [5, 10]. Realizing such gains requires data systems to manage many isolated speculative states, not just execute queries efficiently over one state.

Conventional DBMS mechanisms only approximate this access pattern. MVCC, transactions, and savepoints support concurrency and rollback within one evolving state; physical clones provide isolated copies at coarse granularity. None directly supports persistent, named, independently mutable states that can be revisited, compared, and pruned throughout long-running exploration.

Recent relational systems address this gap by enabling branch operations over database state. Neon shares WAL-derived page histories [8]; TigerData uses copy-on-write distributed block storage [4]; Xata implements block-level copy-on-write below PostgreSQL [9]; and Dolt stores versions in a content-addressed Merkle/prolly-tree engine [7]. These systems expose related branch abstractions but differ along dimensions such as metadata management, compute allocation, garbage collection, copy-on-write granularity, and diff/merge support.

The relevant question is therefore not whether a DBMS exposes branching, but whether its design matches a workload. Concrete agents may explore schema-changing branches with backfills, replay alternative transaction prefixes, or build deep speculative trees with hundreds of states. Even within one domain, workloads can differ in fanout, depth, mutation mix, read intensity, and pruning behavior, so a useful benchmark must parameterize the shape and lifecycle of speculative state.

Existing database benchmarks instead operate over one logical database state. TPC-C and TPC-H define transactional and analytical workloads, while CH-benCHmark combines them for hybrid OLTP/OLAP execution [1]. YCSB and OLTP-Bench parameterize request mixes and OLTP workloads, but both run against a single evolving database instance rather than a tree-structured set of branch states [2, 3]. Similar to SEQUOIA 2000, which exposed how scientific workloads exceeded prior benchmark assumptions [6], agentic speculation needs a benchmark that makes branching structure first-class.

We present BranchBench[1], an extensible benchmark framework for branchable relational DBMSes under agentic workloads. BranchBench instantiates a common branch–mutate–evaluate–prune loop with parameters for worker count, branch fanout and depth, schema mutations, data queries, cross-branch comparisons, and pruning. Users can add backends behind a small branch API and add workloads via parameter configurations, schemas, and SQL templates.

BranchBench is not meant to declare a universal winner among branchable databases or to exhaust all agentic behavior. It provides a

CAIS'26 SAO Workshop, May 26, 2026, San Jose, CA

[1] Benchmark code: https://github.com/ElaineAng/db-fork.

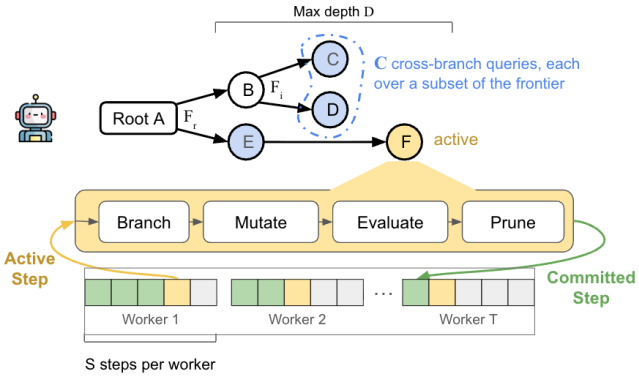

Figure 1: The BranchBench macrobenchmark executes a parameterized branch–mutate–evaluate–prune loop over a shared branch tree.

structured way to express, scale, and compare branching workloads. We make three contributions:

- We define agentic database branching as a benchmark target and identify workload dimensions that are absent from conventional relational benchmarks.
- We design BranchBench, an extensible macrobenchmark and microbenchmark framework that makes branching workloads reproducible and comparable through explicit topology, mutation, evaluation, and pruning parameters.
- We run BranchBench on Neon and Dolt, two branchable DBMSes with contrasting architectures, and show how the benchmark attributes end-to-end differences to branch lifecycle costs, branch-local SQL costs, storage behavior, and resource limits.

2 BENCHMARK FRAMEWORK DESIGN

BranchBench separates the benchmark into a reusable execution engine, benchmark configurations, and backend adapters. This structure is intended to make the benchmark extensible along two axes: users can add new workload configurations by changing parameters, schema files, and SQL templates, and can add new systems by implementing the backend branch API; thus, the benchmark does not prescribe a fixed database schema; the schema is part of each benchmark configuration.

Execution model. Figure 1 shows the macrobenchmark driver. A run starts from a root branch and launches T workers executing concurrently. Each worker executes S steps over a shared branch tree. Workers follow the same step structure and parameter counts, but may choose different parent branches.

Tree shape is enforced during the branch phase: a worker may select only an eligible parent whose child count is below its fanout limit and whose depth is below D. The root fanout F_r, inner fanout F_i, and maximum depth D therefore constrain where new branches can be created rather than changing the step sequence itself. These parameters cover flat stars, wide shallow trees, bushy trees, and deep search trajectories.

Each step has four phases. In the *branch* phase, the worker selects an eligible parent, creates a child branch, and connects to it. In the *mutate* phase, it applies M_s schema mutations and M_d data mutations. In the *evaluate* phase, it executes Q_v read queries that validate or score the branch state. In the *prune* phase, it deletes the branch with probability γ. A branch that completes the step and is not pruned becomes committed; committed leaves form the frontier. A configuration may also include C cross-branch queries over the frontier. These queries are scheduled across the run, are not counted as part of any worker's S steps, and capture comparisons across candidate branch states or aggregate outcomes across simulations.

Table 1 summarizes the benchmark configurations used in our evaluation. The goal is not to restrict agentic speculation to this specific set of parameters; rather, these configurations demonstrate how a user can translate a plausible agentic exploration pattern into a concrete BranchBench benchmark instance. For example, the table includes moderate branching with schema evolution (Software Dev), flat replay branches with heavy data operations (Failure Repro), alternative data-cleaning branches (Data Cleaning) compared by validation queries, deep search trees (MCTS), and wide short-lived fanout patterns (Monte-Carlo Simulation).

Backend interface and microbenchmarks. BranchBench abstracts each backend behind timed operations for **branch creation**, **branch connection**, **branch deletion**, and **SQL execution**. Both macrobenchmarks and microbenchmarks use this interface, so measurements are comparable across end-to-end workflows and isolated operation studies. Macrobenchmarks compose these operations into full branch–mutate–evaluate–prune executions and report completion time, branch-management fraction, and storage overhead. Microbenchmarks reuse the same operations in scripted setup and timed phases to isolate branch lifecycle costs, data and schema operations, and single- or multi-threaded execution effects. This shared instrumentation lets BranchBench attribute workflow-level slowdowns to specific measured components, such as branch lifecycle latency, schema mutation cost, read-query latency, or contention under concurrent execution.

3 EVALUATION ON NEON AND DOLT

We run BranchBench on Neon and Dolt because they were mature enough to run our representative branchable-relational workloads; other systems we considered, including Xata and TigerData, were not yet mature enough for our macrobenchmarks due to constraints such as high branch latency and small branch limits. Our goal is therefore not to rank all branchable databases, but to show that common workload definitions can expose architecture-specific bottlenecks.

We use the most usable deployment mode available for each system. Despite non-trivial effort, we could not run Neon's open-source version reliably, so we evaluate Neon through its hosted cloud service in `us-east-1`. Dolt could be built and run from source, so we run a source-built Dolt server on a `c8i.4xlarge` EC2 instance in `us-east-1`. This is not a controlled apples-to-apples deployment: Neon includes hosted control-plane and network effects, while Dolt runs in a controlled EC2 environment. This affects absolute latencies, but not our purpose of attributing bottlenecks: the dominant effects we observe remain visible despite point network latency. A system owner can remove this artifact by running BranchBench in a fully controlled environment.

Group	Parameter	Software Dev	Failure Repro	Data Cleaning	MCTS	MC Simulation
Setup	Workers T	5	1	10	10	1000
	Steps/worker S	20	10	20	100	1
	Cross-branch C	1	—	2	—	1
	Root fanout F_r	5	10	10	10	1000
	Inner fanout F_i	3	—	3	10	—
	Depth D	3	1	3	25	1
Per-step	Schema changes M_s	1	5	1	—	—
	Data mutations M_d	1	45	1	1	50
	Read queries Q_v	2	1	1	1	1
	Prune prob. γ	0.1	1	—	0.1	1

Table 1: Representative BranchBench macrobenchmark instantiations. Setup parameters define the run shape and scale; per-step parameters define the work performed at each step; — means the operation is absent.

System	Software Dev	Failure Repro	Data Cleaning	MCTS	MC Simulation
Neon	~	✓	~	~	✗
Dolt	✓	✓	✓	~	✗

Table 2: Full-workflow capability for the two demonstration systems within a two-hour timeout. ✓indicates completion, ✗indicates inability to execute the workload, and ~indicates partial execution or timeout.

System	Software Dev	Failure Repro	Data Cleaning	MCTS	MC Simulation
Neon	4.0 GB	0 B	4.7 MB	4.1 MB	—
Dolt	93.1 MB	63.6 MB	94.1 MB	3.8 MB	—

Table 3: Storage overhead per workflow for Dolt and Neon.

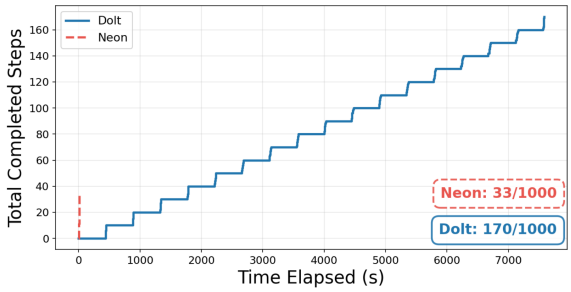

Figure 2: MCTS progress before timeout with full . With a total of 1000 steps, Neon finished 33 steps before stalling at the live-branch limit; Dolt slowly finished 170 steps before timeout.

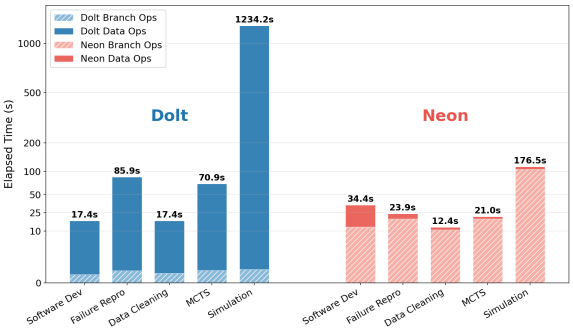

Figure 3: End-to-end latency by workflow on the mini configuration.

We instantiate two workload scales over the CH-benCHmark schema [1]. The full configuration uses five warehouses and the parameters in Table 1; Appendix A gives representative SQL templates. This configuration tests whether each system can support the intended branch topology and lifecycle at scale. Because neither system completed every full workflow within a two-hour timeout, we also use a mini configuration with one warehouse and reduced worker and step counts. The mini configuration preserves each workflow's per-step operation mix, allowing both systems to complete comparable runs for bottleneck attribution.

Full macrobenchmark statistics. Table 2 reports the full configuration, where BranchBench checks whether each system can execute the intended branch lifecycle and topology at scale. Neither system completes the full suite. Neon completes Failure Reproduction, but only partially executes Software Development, Data Cleaning, and MCTS because its hosted deployment limits the number of concurrent live branches. In our experiments, Neon reached a 20-active-branch limit even on its highest subscription tier; although documented as a soft limit, such caps are a practical constraint for workloads with many live speculative states. Dolt completes Software Development, Failure Reproduction, and Data Cleaning, but MCTS and Simulation become dominated by branch-local reads.

Figure 2 shows why both branch metadata and branch-local data operations matter for progress: Neon advances quickly until hitting the live-branch cap, while Dolt continues branching but slows on data-intensive joins.

Storage behavior. Table 3 reports storage at workflow end or timeout. The same workload definitions expose different storage behavior: Neon uses over 43× more storage on Software Dev because backfilled schema updates reduce page sharing, while Dolt uses more storage on Failure Repro and Data Cleaning because it retains pruned-branch history and materializes `DEFAULT FALSE` columns. Note that Neon's storage values should be treated as order-of-magnitude evidence only: Neon and Dolt did not always

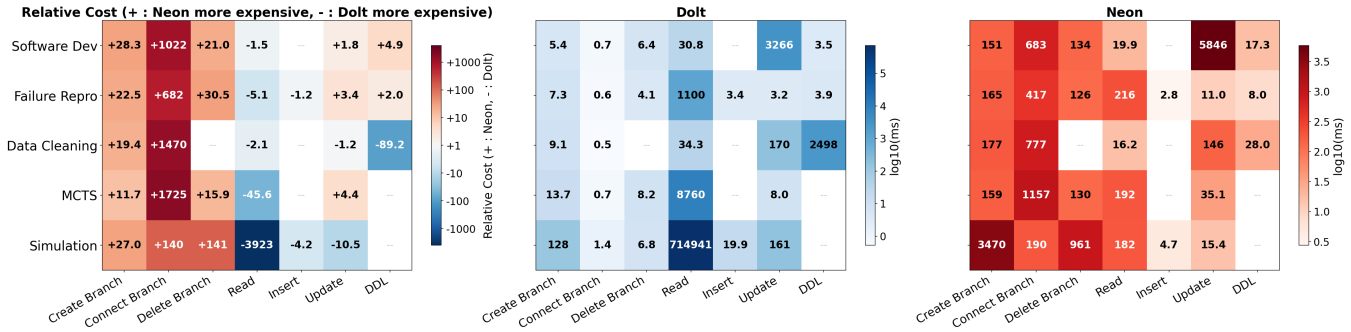

Figure 4: Median operation latency for Neon and Dolt on mini macrobenchmark configurations.

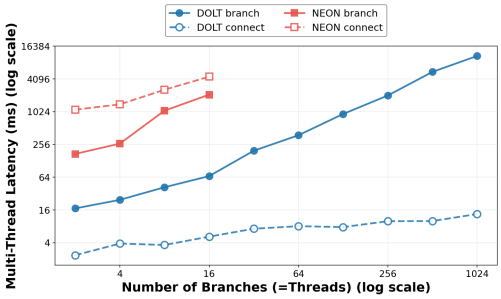

Figure 5: Branch creation and connection latency when creating x branches with x threads.

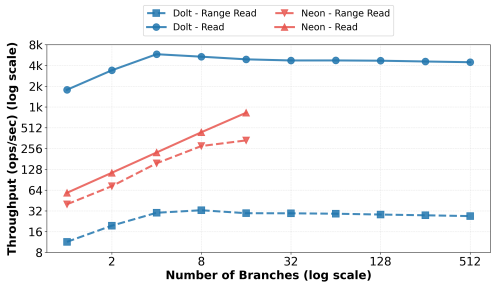

Figure 6: Read capacity as the number of branches increases (range size = 100).

complete the same number of steps before timeout, and Neon's hosted metrics start after a delay and update only periodically. This means Neon may under-report storage for workflows that create and prune branches; Dolt was measured directly from the local database directory.

Mini macrobenchmark statistics. The mini macrobenchmarks let Neon and Dolt complete the same workloads. Figure 3 reports end-to-end latency by workflow, and Figure 4 decomposes latency by operation class. Across workflows, Neon spends most time in branch management, while Dolt spends most time in branch-local data operations. Neon executes branch-local SQL faster on these workloads, but branch management is slower; Dolt branches cheaply, but range reads and joins become 5–4000× slower than Neon's, despite not paying hosted-network round trips. DDL also diverges: Dolt materializes default values during data-cleaning schema changes, whereas PostgreSQL-native engines treat them as metadata operations.

Microbenchmark breakdown. Microbenchmarks separate branch-management cost from branch-local data-access cost. For branch management, Figure 5 measures creating and connecting to x branches with x client threads: Neon's latency is substantially higher because each branch must be registered through the hosted control plane and attached to branch-specific compute, whereas Dolt implements branch creation and checkout by updating local metadata over versioned root hashes. For branch-local data-intensive queries, Figure 6 measures range-read throughput as branches increase: Neon's per-branch compute improves aggregate read capacity until the live-branch limit, while Dolt's shared process and prolly-tree traversal make throughput plateau. These drill-downs show how BranchBench turns a coarse comparison into attributable measurements; the microbenchmark suite also supports range updates, deletes, DDL, and new operation templates.

4 CONCLUSION

BranchBench provides a benchmark framework for making agentic database branching reproducible and comparable. It turns speculative exploration into explicit workload dimensions: branch topology, lifecycle operations, branch-local SQL, cross-branch comparison, and pruning. Its macrobenchmarks test whether a system can execute complete branch–mutate–evaluate–prune workflows, while its microbenchmarks isolate the primitive costs needed to explain end-to-end behavior. Running BranchBench on Neon and Dolt illustrates the benchmark's diagnostic role: the same workload definitions expose different limits and reveal trade-offs between branch agility and branch-local query performance. More broadly, BranchBench gives system users and builders a concrete way to compare branchable database designs against the speculative workloads they are meant to support.

REFERENCES

[1] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Anisoara Patel, Meikel Poess, Kai-Uwe Sattler, Bernhard Seeger, Jan Takahashi, and Marek Wolski. 2011. Mixed Workload CH-benCHmark. In *Proceedings of the Fourth*

International Workshop on Testing Database Systems (DBTest). https://doi.org/10.1145/1988842.1988850
[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*. https://api.semanticscholar.org/CorpusID:2589691
[3] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7 (2013), 277–288. https://api.semanticscholar.org/CorpusID:2612270
[4] Mike Freedman. 2025. Fluid Storage: Forkable, Ephemeral, and Durable Infrastructure for the Age of Agents. https://www.tigerdata.com/blog/fluid-storage-forkable-ephemeral-durable-infrastructure-age-of-agents.
[5] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. arXiv:2601.11868 [cs.SE] https://arxiv.org/abs/2601.11868
[6] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. 1993. The SEQUOIA 2000 storage benchmark. , 10 pages. https://doi.org/10.1145/170036.170038
[7] Dolt Team. 2025. Dolt - Using Branches. https://docs.dolthub.com/sql-reference/version-control/branches.
[8] Neon Team. 2026. Neon Branching - Branch your data the same way you branch your code. https://neon.com/docs/introduction/branching.
[9] Xata Team. 2025. Xata Instant Branching: Copy-on-Write branching system for PostgreSQL databases. https://xata.io/documentation/core-concepts/branching.
[10] Jiakai Xu, Tianle Zhou, Eugene Wu, and Kostis Kaffes. 2025. Toward Systems Foundations for Agentic Exploration. *SOSP SSA Workshop* (2025).

A MACROBENCHMARK QUERIES

This section provides examples of the queries used in the macrobenchmark.

A.1 Software Development

Schema Change:
```
ALTER TABLE customer ADD COLUMN loyalty_tier_xx;
```
Data Mutation (backfill):
```
UPDATE customer SET loyalty_tier_xx = (CASE WHEN ...);
```
Read (evaluate quality):
```
SELECT loyalty_tier_xx, COUNT(*) FROM customer GROUP BY loyalty_tier_xx;
```
Compare (cross-branch):
```
SELECT loyalty_tier_xx, COUNT(*) AS tier_count,
AVG(c_ytd_payment) AS avg_payment
FROM customer GROUP BY loyalty_tier_xx;
```

A.2 Failure Reproduction

Schema Change (replay):
```
ALTER TABLE order_line ADD COLUMN X;
ALTER TABLE orders ADD/DROP COLUMN Y;
ALTER TABLE customer ADD COLUMN Z;
```
Data Mutation (replay):
```
INSERT INTO orders (...);
UPDATE customer SET (...);
DELETE FROM order_line WHERE (...);
```
Read (invariant check):
```
SELECT DISTINCT o.o_id FROM orders
JOIN order_line ON (...)
GROUP BY (...) HAVING (...);
```

A.3 Data Cleaning

Schema Change (aux column):
```
ALTER TABLE customer ADD COLUMN c_clean_xx DEFAULT FALSE;
```
Data Mutation (strategies):
```
UPDATE customer SET c_balance = 0 WHERE c_balance IS NULL;
DELETE FROM customer WHERE c_balance IS NULL;
```
Read (correctness):
```
SELECT COUNT(CASE WHEN c_balance < 0 THEN 1 END)
AS invalid FROM customer;
```
Compare (cross-branch):
```
SELECT COUNT(...) AS invalid,
MAX(c_ytd_payment)-MIN(c_ytd_payment) AS spread
FROM customer;
```

A.4 Monte Carlo Tree Search

Data Mutation:
```
UPDATE stock SET s_quantity = (...) WHERE (...);
```
Read (expected reward):
```
SELECT SUM(ol_amount) AS total_cost
FROM order_line JOIN warehouse ON (...);
```

A.5 Simulation

Data Mutation:
```
INSERT INTO orders (...);
UPDATE stock SET (...) WHERE (...);
INSERT INTO order_line (...);
```
Read (evaluate outcomes):
```
SELECT SUM(...) AS stockouts,
SUM(ol.ol_amount) AS total_cost
FROM stock JOIN order_line ON (...);
```
Compare (aggregate across branches):
```
SELECT AVG(stockouts), AVG(total_cost)
FROM all_branches.outcome_metrics;
```