Vive la Différence: Practical Diff Testing of Stateful Applications

Kexin Zhu
Google
zhukexin@google.com

Michael Whittaker
Google
mwhittaker@google.com

Srdjan Petrovic
Google
spetrovic@google.com

Robert Grandl
Google
rgrandl@google.com

Sanjay Ghemawat
Google
sanjay@google.com

ABSTRACT

Software rollout is the process of replacing the version of an application that is currently running in production with a new version. Many subtle and catastrophic bugs occur during software rollout. There are many existing techniques to improve the odds of a rollout completing successfully, but these techniques don't work well when the application has shared, persistent, mutable state. In this paper, we present a practical framework to test the rollout of stateful applications. Our framework uses diff testing to verify that the new version of an application behaves identically to the currently running version that will be replaced. The framework has three main components to safely and efficiently compare the behavior of the two versions. First, we implement database branching on top of Postgres. Second, we implement an efficient algorithm to diff two database branches. Third, we describe how to replay client requests to improve test coverage. Finally, we identify three common categories of rollout bugs and demonstrate how our framework can find these bugs with minimal performance overhead.

PVLDB Reference Format:
Kexin Zhu, Michael Whittaker, Srdjan Petrovic, Robert Grandl, and Sanjay Ghemawat. Vive la Différence: Practical Diff Testing of Stateful Applications. PVLDB, 18(7): 2018 - 2030, 2025.
doi:10.14778/3734839.3734841

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ServiceWeaver/database.

1 INTRODUCTION

Software *rollout* is the process of replacing the version of a system that is currently running in production with a new version, without disrupting the serving of live production traffic. Software rollout is an error-prone process that forces you to anticipate, reason about, and test all possible interactions between the old and new versions of your system. Rolling out *stateful systems*—*i.e.* systems with shared, persistent, mutable state—is even more complex because of the state that persists across versions.

Unsurprisingly, many bugs occur during the rollout of stateful systems [32]. Based on our analysis of bugs reported within Google, we found that stateful rollout bugs can impact many services, require significant time and effort to mitigate, and can lead to substantial revenue loss.

Existing rollout and testing techniques struggle to detect and mitigate these bugs. Canarying [2], rolling updates [26], and blue/-green deployment [12] are widely used techniques that gradually deploy a new software version to a subset of users to limit the blast radius in case the new version is buggy. However, these techniques don't work as well when applied to stateful systems. Because of shared state, a bug in one application version can easily snowball into a bug across all versions, leading to a large bug blast radius. Standard testing techniques like unit tests, end-to-end tests, and integration tests are effective at catching bugs in a single application version, but they don't typically test the interactions between multiple software versions that occur during a rollout.

In this paper, we present a framework for testing the rollout of stateful applications. Concretely, our framework implements a version of traditional *diff testing* [28] that is generalized to test stateful applications that interact during a software rollout. Before rolling out a new version of a system v_2 to replace the version v_1 that is currently running in production, our framework executes a number of client requests against both v_1 and v_2 to ensure that there are no unexpected differences in their behavior. To accomplish this, our framework solves the following technical challenges.

- C1: How do we compare the two versions, v_1 and v_2, on production data without harming the production deployment?
- C2: How do we test the subtle interactions that occur between v_1 and v_2 when requests are interleaved between both versions during a rollout?
- C3: How do we capture the changes v_1 and v_2 make to persistent state, and how do we reason about whether these changes are intended or buggy?

Different solutions to address each challenge in isolation have been proposed both by academia and industry. For example, there exist versioned databases [6, 11, 14, 16, 27], diffing techniques [5, 13], and tools to populate test databases [3]. Godefroid *et al.* [9] present diff testing solutions for REST APIs, without providing a solution for computing differences between two database instances. In this paper, we present a diff testing framework that addresses all of the challenges above.

- We present an efficient database branching layer that allows us to quickly execute a large number of requests against a production-scale database without impacting the actual production database.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734841

- We present an efficient way to compute the difference between two database branches that is independent of the size of the underlying database. We also present a way to display these differences using a three-way relational format that is easy for developers to understand.
- We present a method to diff test entire execution sequences and interleave requests between two software versions in order to catch subtle bugs.

We implement these ideas on top of Postgres [22]. Branches can be created almost instantly (< 100ms), and the overhead of reading from and writing to a branch is modest. Our solution is able to quickly identify the types of bugs we've seen in production and can provide to the user output that helps them quickly understand the root causes of the bugs.

2 BUG ANALYSIS

We begin by describing the kinds of bugs our testing framework aims to catch. We surveyed software rollout bugs that were reported inside Google and in open-source projects [32]. We found three main categories of bugs: *data corruption*, *data incompatibility*, and *false data assumptions*. These bugs went unnoticed for days to weeks, and teams spent days to months fixing them and developing tools to prevent future occurrences. These bugs caused many internal services to be impacted, leading to substantial revenue loss. We now provide examples of these three types of bugs that occurred at Google. As we show in Section 8, our solution is able to catch these bugs and enables developers to quickly fix them before they arise in production.

Data Corruption. This type of bug involves corrupted data in a database. Data corruption bugs are especially troublesome because corrupted data can lie dormant in a database for a long period of time before suddenly causing widespread impact. We found that it is common for corrupted data to be present in production databases for days before being discovered and reported.

One example in this category was a bug caused by protobuf [25] serialization. In rare cases, serialized protobufs were being truncated before being written to the database. This bug went unnoticed for 10 days, causing one known database corruption and many internal services being impacted. To find all the corrupted data, the team spent a month developing a data corruption detection tool.

Data Incompatibility. This type of bug involves two versions v_1 and v_2 of a system that have inconsistent interpretations of the same data. For example, the two versions might disagree on the endianness of some piece of encoded data. Data incompatibility bugs are easily missed in testing because versions v_1 and v_2 are often correct in isolation and are only buggy when they interact.

One example in this category was a bug caused by a version v_1 of a system that assumed a column in a relational database was always empty. Version v_2 was rolled out to "fix" the problem and allow the column to be non-empty. v_2 began to populate the column which caused v_1 to start erroneously dropping rows. This bug went unnoticed for 5 days. The bug had serious production impact causing hundreds of rows to be dropped from the production database. It took half a day to mitigate the incident. A postmortem of the incident cited a lack of data compatibility testing as a key contributor to the outage.

False Data Assumptions. This type of bug involves an application version with incorrect assumptions about the data it accesses. For example, a system may erroneously assume that an integer-valued column in a database is always non-negative. This type of bug can be hard to catch if the system is tested against data that is not representative of the data in production. Testing against unrepresentative data is common. If a developer writes code with faulty assumptions, they will likely craft synthetic data for testing that reflects those same faulty assumptions.

One example of a bug in this category was caused by a system with a hard-coded upper limit on the expected size of a database table. A new version of the system was rolled out that lowered this limit. Because the new version was consistently tested on much smaller databases, the lower limit appeared unproblematic. However, the size of the production database exceeded this limit, causing jobs to crash in production. This bug was detected more than 20 minutes after the production outage began. It took several hours to identify the root cause and one day to resolve it, resulting in substantial revenue loss across different products.

3 DIFF TESTING

In this section, we provide background on diff testing. In later sections, we explain how our testing framework leverages diff testing.

Differential testing (diff testing) was first introduced by McKeeman [15] to complement traditional software testing processes. Over the past decade, diff testing has been quickly adopted by industry as a means of making testing processes more robust [5, 10, 15, 17]. We begin by explaining the diff testing of *stateless* systems. Then, we discuss the challenges of diff testing *stateful* systems.

3.1 Stateless Diff Testing

Diff testing is a testing technique to find unexpected differences in the behavior of two pieces of code that are supposed to behave identically. For example, consider the two functions in Figure 1 which return the *n*th Fibonacci number.

```
def fib1(n):                def fib2(n):
  if n <= 1:                  x, y = 0, 1
    return n                  for _ in range(n):
  return (fib1(n-1) +           x, y = y, x + y
          fib1(n-2))          return x
```

Figure 1: Two (identical?) Fibonacci implementations

A diff test of these two functions repeatedly selects a value for n and evaluates `fib1`(n) and `fib2`(n), trying to find a case where `fib1`$(n) \neq$ `fib2`(n). For example, `fib1`$(-1) = -1$, while `fib2`$(-1) = 0$. The exact procedure by which inputs are generated is out of scope for this paper. Typically, they are generated randomly or drawn from a pool of hand-crafted inputs.

While diff testing can be used to test individual functions—like those in Figure 1—it is most commonly used to test entire versions of an application. Specifically, diff testing can test that it is safe to roll out a version v_2 of a system to replace an existing version v_1 that is serving production traffic. We assume that when a version of the system v receives a client request r, it replies with a response $v(r)$. Note that the exact format of the requests and responses is immaterial (e.g., HTTP, RPC, REST, etc.). As described above, diff testing attempts to find a request r such that $v_1(r) \neq v_2(r)$. If v_1 and v_2 respond differently to the same client request, it is a signal that v_2 might be buggy and unsafe to roll out.

Not all differences in behavior between v_1 and v_2 are a sign that something is buggy. In fact, v_2 often *intentionally* behaves differently than v_1. For example, v_2 might introduce a new feature to v_1 or fix an existing bug in v_1. To determine whether a difference in behavior between v_1 and v_2 is cause for alarm, diff testing frameworks report diffs to developers. If $v_1(r) \neq v_2(r)$, a diff of responses $v_1(r)$ and $v_2(r)$ is shown to the developer to validate if the difference is intended. Developers can manually inspect the diffs or write filters to automatically validate diffs. The exact mechanism by which diffs are processed is out of scope for this paper.

3.2 Stateful Diff Testing

Consider a version v of a *stateful* system that receives client request r and has access to a database in some initial state σ. It outputs $o = v(r)$ and also potentially mutates the database to a new state σ'. To diff test two versions v_1 and v_2, we want to find a request r and database state σ such that the versions produce different outputs (*i.e.* $v_1(x) \neq v_2(x)$) or yield different database states (*i.e.* $\sigma_1 \neq \sigma_2$). This is illustrated in Figure 2.

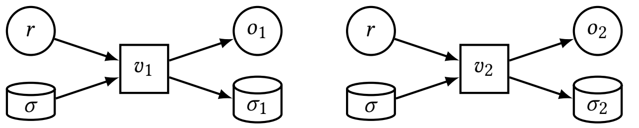


Figure 2: An illustration of *stateful* diff testing. The client request r is processed by software versions v_1 and v_2 with initial database state σ. This produces responses o_1 and o_2 and database states σ_1 and σ_2, which are diffed.

Note that the diff testing process illustrated in Figure 2 must allocate a separate database instance for each application version v_1 and v_2. If the two versions instead shared the same database instance, request r would be executed twice on the database: once by v_1 and once by v_2. This double execution is not only incorrect, but also makes it impossible to distinguish the changes made by v_1 from the changes made by v_2. Therefore, for every request, each version is given its own isolated database instance to mutate. This way, we end up with two independent database states (*e.g.*, σ_1 and σ_2 in Figure 2) that we can diff.

As with stateless diff testing, developers sometimes deliberately make changes to v_2 to make it behave differently than v_1. Thus, if a diff testing framework finds a request r that, when executed by versions v_1 and v_2, yields unequal database states σ_1 and σ_2, the diff testing framework presents the diff of the two database states to the developer. The developer is responsible for inspecting the diffs and determining if they are intended or not.

4 FRAMEWORK OVERVIEW

Our stateful diff testing framework takes the following three inputs:

(1) **An initial database state** σ. Our framework is agnostic to how this database state is produced. For example, it can be an empty database, or a synthetic database, or a snapshot of the production database. However, we recommend that the production database is copied into a testing environment, as shown in Figure 3a.
(2) **A set of client requests** $r_1, r_2, \ldots, r_n$. Our framework is agnostic to how these requests are produced. They can be sampled from production traffic, manually written, or randomly generated.
(3) **Two binaries** v_1 **and** v_2. Typically, v_1 is currently running in production, and v_2 is the binary that will replace v_1.

Given these inputs, the framework performs a sequence of steps grouped into three sequential stages: *Branching*, *Replaying*, and *Diffing*. The branching stage creates multiple branches of the initial database state σ, as illustrated in Figure 3b. These branches are like the branches you are familiar with in version control systems like git [7]. They are lightweight, isolated, readable and writable views derived from the initial database state. Notably, the branches are *not* full copies of the database. They are cheap to create and small in size. Section 5 describes branching in more detail.

The replaying stage then executes each request r against binaries v_1 and v_2, as illustrated in Figure 3c. Each binary is executed against one of the branches. This produces outputs o_1 and o_2 from v_1 and v_2, respectively. It also leaves the branches in states σ_1 and σ_2 produced by the modifications from v_1 and v_2, respectively. Section 6 describes the replaying stage in more detail.

The diffing stage computes differences between o_1 and o_2 as well as σ_1 and σ_2. If the diffs are non-empty, they are shown to the developer. Section 7 describes the diffing stage in more detail.

5 BRANCHING

Branching is the process of creating a view of the initial database that is lightweight, isolated, readable, and writable. Our framework relies on branching to quickly create multiple copies of the initial database. These branches are then used as the underlying storage for replaying (Section 6) and diffing (Section 7).

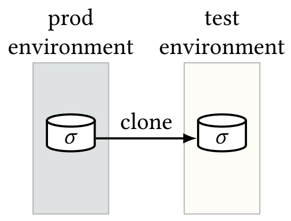
As discussed earlier, branching needs to be a fast and lightweight operation. In particular, creating physical *copies* of the underlying database is not a realistic solution. Likewise, copying the production database state into a different database system that supports branching (e.g., Dolt [6] or Neon [16]) isn't recommended either, both for performance reasons and for the fact that the destination database system is likely to have slightly different SQL semantics than the original database.

For this reason, we implement a "bolt-on" approach for our diff testing framework, built on top of Postgres triggers. Our implementation can efficiently branch Postgres databases without requiring any modifications to the Postgres source code or to the application being tested. It also avoids the need for complex query rewrite. In this section, we describe the design of our "bolt-on" solution, and in Section 8, we evaluate its performance.

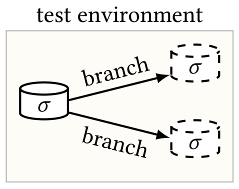
5.1 Branches

Consider a Postgres database with relation R. To create a branch of R, we create two auxiliary tables R^+ and R^- as well as a database view R', all of which have (approximately) the same schema as R.

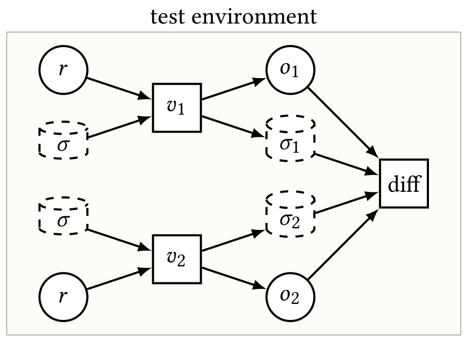
- R^+ contains the insertions into R'.



(a) The production database is cloned from the production environment into a testing environment where the diff testing will be performed.



(b) For every request r, two branches of the database are created. The branches are lightweight, isolated, readable and writable views.



(c) After the branches are created, the request r is run through versions v_1 and v_2. Each version executes against its own branch. The final outputs (o_1 and o_2) and final database branch states (σ_1 and σ_2) are diffed.

Figure 3: An illustration of the execution of our stateful diff testing framework

a	b
1	1
2	2

R

a	b

R^+

a	b

R^-

a	b
1	1
2	2

R'

(a) The initial contents of R, R^+, R^-, and R'.

a	b
1	1
2	2

R

a	b
3	3

R^+

a	b

R^-

a	b
1	1
2	2
3	3

R'

(b) To insert the tuple $(3, 3)$ into the branch, the tuple is inserted into R^+. R' reflects this insertion.

a	b
1	1
2	2

R

a	b
3	3

R^+

a	b
2	2

R^-

a	b
1	1
3	3

R'

(c) To delete the tuple $(2, 2)$ from the branch, the tuple is inserted into R^-. R' reflects this deletion.

a	b
1	1
2	2

R

a	b
3	3
1	42

R^+

a	b
2	2
1	1

R^-

a	b
1	42
3	3

R'

(d) To update the tuple $(1, 1)$ to $(1, 42)$ in the branch, the tuple $(1, 1)$ is inserted into R^-, and the tuple $(1, 42)$ is inserted into R^+. R' reflects this update.

Figure 4: An example insertion (4b), deletion (4c), and update (4d) of a branched relation $R(a, b)$. Notable rows are in yellow.

- R^- contains the deletions from R'.
- R' contains the final contents of the branch and is defined as $R' = R + R^+ - R^-$, where + and − are multiset operations.
- Reads are performed against R'.

This branching scheme is best explained through an example. Suppose we have a table R defined as follows.

```
CREATE TABLE R(a int PRIMARY KEY, b int);
```

To branch R, we create tables R^+ and R^-. Note that R^+ and R^- have the same schema as R except without the primary key constraint, so they provide multiset semantics. We also create a view $R' = R + R^+ - R^-$.

```
CREATE TABLE RPlus(a int NOT NULL, b int); -- R+
CREATE TABLE RMinus(a int NOT NULL, b int); -- R−
CREATE VIEW RPrime AS (SELECT * FROM R) -- R′
UNION ALL (SELECT * FROM RPlus)
EXCEPT ALL (SELECT * FROM RMinus);
```

Imagine that R has contents $\{(1, 1), (2, 2)\}$, as illustrated in Figure 4a. R^+ and R^- are initially empty, and R' is expectedly equal to R. We now walk through an example of what happens when we insert a tuple into the branch, delete a tuple from the branch, and update a tuple in the branch.

- **Insertion**. To insert the tuple $(3, 3)$ into the branch, the tuple $(3, 3)$ is inserted into R^+, as illustrated in Figure 4b. Note that R is unmodified, and R' correctly reflects the insertion of the tuple $(3, 3)$.
- **Deletion**. To delete the tuple $(2, 2)$ from the branch, the tuple $(2, 2)$ is inserted into R^-, as illustrated in Figure 4c. Note that R is again unmodified, and R' correctly reflects the deletion of the tuple $(2, 2)$. Note that the tuple $(2, 2)$ is added to R^- only if

the tuple already exists in R'. We discuss the details of how we accomplish this momentarily.

- **Update**. To update the tuple $(1, 1)$ to $(1, 42)$ in the branch, the tuple $(1, 1)$ is inserted into R^-, and the tuple $(1, 42)$ is inserted into R^+, as illustrated in Figure 4d. The update is effectively treated as the deletion of tuple $(1, 1)$ followed by the insertion of the tuple $(1, 42)$.

We've described how to branch a single relation. To branch an entire database, we simply branch every relation in the database.

5.2 Queries

Insertions into a branch should be written to R^+; deletions should be written to R^-; and updates should be written to both. We now explain how we intercept queries to accomplish this. For example, suppose a binary (v_1 or v_2) executes the following SQL statement which deletes rows from R:

```
DELETE FROM R Where b < 2;
```

When executing against a branch of R, this statement should *not* delete rows from R. Instead, our diff testing framework needs to find all rows in R' that satisfy the condition `b < 2` and then insert these rows into R^-.

First, we need to adjust queries to replace all references to a relation R with references to the branched view R'. We can do this without modifying the developer's binary and without rewriting any queries by leveraging some renaming trickery. When branching a relation R, we rename it R_{old}, and then name the branched view R instead of R'. This way, all existing queries in the developer's binary target the branched view rather than the underlying table.

Second, we need to intercept queries issued against the branched view. To achieve this, we leverage Postgres' support for registering triggers on views [23]. Triggers on views are a special type of database trigger that allow you to define actions to be performed in response to modifications made to a view, even if the view itself isn't directly modifiable. Triggers on views work by intercepting data modification operations (`INSERT`, `UPDATE`, `DELETE`) targeted at the view and then executing predefined actions instead of directly modifying the view's data. While our framework requires Postgres, triggers on views are supported by multiple other relational databases (e.g., Oracle [18], Microsoft SQL Server [20]).

For example, the trigger that translates deletions from R' to insertions into R^- is shown in Figure 5. When a tuple is deleted from R', the tuple is bound to the variable `OLD`, and the `R_deletion` function is executed, which inserts `OLD` into R^-. The trigger is only executed on tuples that exist in R'. Attempting to delete a tuple that is not present in R' will not invoke the trigger. The triggers for handling insertions and updates are similar. Our framework generates these triggers automatically when a branch is created.

5.3 Constraints

Constraints are rules enforced on the data in a relational database to maintain its integrity, accuracy, and consistency. `PRIMARY KEY`, `FOREIGN KEY` and `NOT NULL` are very common constraints. When our diff testing framework branches a relation R with constraints, it generates a trigger that enforces the relation's constraints. For example, the trigger for handling insertions into a relation R with a primary key column a asserts that there does not already exist a tuple in R' with $a = R'.a$. Our diff testing framework currently supports primary key, unique, non-null, and foreign key constraints.

```
CREATE OR REPLACE FUNCTION R_deletion()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS BEGIN
    INSERT INTO RMinus(a, b)
    VALUES (OLD.a, OLD.b);
    RETURN OLD;
END;
```

Figure 5: A trigger that translates deletions from R' to insertions into R^-.

5.4 Limitations

While our trigger-based "bolt-on" approach to branching is simple, easy to adopt, and covers the common case, it has limitations. There are some relations we cannot branch, and our branched databases do not have 100% feature parity with plain databases. For example, we do not support `CHECK` constraints or relations with preexisting triggers. If an application depends on one of these unsupported features, it cannot be tested with our framework.

These limitations are not fundamental to our approach. We can support more database features. However, each feature requires engineering effort to support. We believe it is possible (and not too onerous) to support the most widely used features, but it would require effort to support the long tail of seldom used features.

6 REPLAYING

Replaying is the second stage in our stateful diff testing framework. It takes as input the branched copies of the database, as well as the set of input requests $r_1, r_2, ..., r_n$, and executes those requests against binaries v_1 and v_2.

Replaying requests on top of a *stateless* system is easy: every request is independent of every other request. That is, for every pair of requests r_1 and r_2, r_1 and r_2 commute, *i.e.* they can be executed in either order or not at all without any influence on future requests.

Replaying requests on top of a *stateful* system is challenging: every request may mutate the database state in a way that affects all subsequent requests. In fact, the entire purpose of persistent mutable state is to enable meaningful interactions between requests. For example, an e-commerce application would be useless if a checkout was independent of prior requests to add items to a cart.

Sequences of requests: For stateful systems, therefore, it is insufficient to test only a single request at a time. Even if every request executes correctly in isolation, a sequence of requests may still exhibit buggy behavior. Thus, our diff testing framework executes sequences of requests, where every request is executed against the database state produced by the previous request, as illustrated in Figure 6. Each request r_i takes as input the preceding database state σ_{i-1} and generates output o_i and a new database state σ_i, across the two binary versions v_1 and v_2. These intermediate outputs and database states are diffed, as described in Section 7.

Interleaving requests: Recall that diff testing is used to build confidence that version v_2 of a system can be safely rolled out

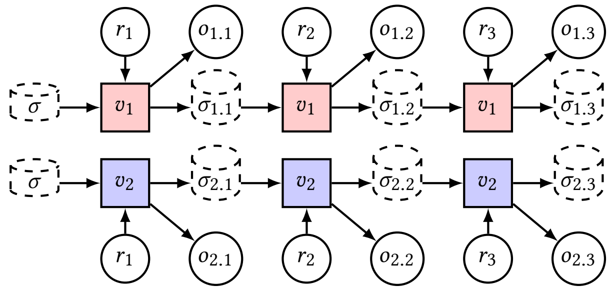


Figure 6: An illustration of diff testing a sequence of requests. Requests r_1, r_2, and r_3 are executed against versions v_1 and v_2. The responses and intermediate database states are diffed.

to replace the existing version v_1. During the rollout of v_2, some requests will be executed by v_1 and others by v_2. It is this interaction between the two versions where bugs often hide [32]. Therefore, our diff testing framework interleaves requests across the two versions v_1 and v_2, as illustrated in Figure 7.

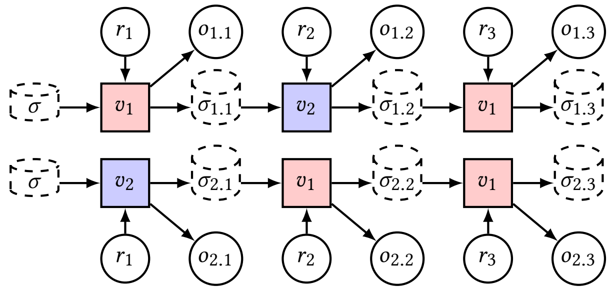


Figure 7: An illustration of interleaved request execution. The requests r_1, r_2, and r_3 are executed in an interleaved fashion against versions v_1 and v_2. Note that these are only two of many possible interleavings. For n requests, there are 2^n total interleavings.

More generally, for every request sequence, our diff testing framework will execute the sequence entirely against v_1, entirely against v_2 (as shown in Figure 6), and interleaved between v_1 and v_2 for some large but bounded number of random interleavings (as shown in Figure 7). This interleaving checks that v_2 is backward-compatible with v_1 and that v_1 is forward-compatible with v_2. Note that every interleaving is executed against its own branch.

Limitations: Our diff testing framework executes all requests serially, so it cannot be used to catch concurrency bugs. This limitation is largely fundamental. If requests are replayed concurrently, their execution is not deterministic. Different executions of the *same* sequence of requests, let alone different sequences of requests, could produce very different client replies and database states. As a consequence, different sequences cannot be meaningfully diffed when executed with concurrency.

7 DIFFING

Diffing is the third and final stage in our stateful diff testing framework. It takes as input the replaying stage output, namely database branches and client replies, and computes differences between them.

Recall from Figure 3 that a request r is executed against versions v_1 and v_2 to produce client replies o_1 and o_2 as well as database states σ_1 and σ_2. If the replies are unequal ($o_1 \neq o_2$) or the database states are unequal ($\sigma_1 \neq \sigma_2$), a summary of the differences is shown to the developer to determine if the differences are expected or unexpected. It is thus critical that these diffs are easy for developers to interpret and understand.

While the existing techniques for diffing *client replies* [5, 13] work well and produce intuitive, easy to understand diffs, database diffing remains challenging. In the rest of this section, we focus on devising a practical database diffing solution.

There are two key aspects to diffing database states: how they are *displayed* and how they are *computed*. The goal of displaying the diffs is to present only the relevant differences to the user, and to do so in a simple and easy to understand manner. This goal is especially important for databases, which are by nature large. Section 7.1 describes our approach to displaying diffs. We present separate solutions for tables with and without primary keys.

As previously mentioned, for a stateful diffing framework to be usable, the process of computing diffs must be efficient. In particular, it is not acceptable for diff computation to iterate over the entire database; instead, it should iterate over only the database portions that have seen changes. Section 7.2 describes our algorithm for computing database diffs, built on top of the branching solution presented in Section 5.

We only describe how to diff *two* database states. We leave the problem of diffing more than two database states to future work.

7.1 Displaying Diffs

7.1.1 With Primary Keys. Consider again the relation $R(\underline{a}, b)$ with primary key column a and non-primary key column b. The initial state, R, of the database is shown in Figure 8a. States R_1 and R_2, produced by v_1 and v_2 after executing some request, are shown in Figure 8b and Figure 8c. Note that v_1 deleted both tuples $(1, 1)$ and $(2, 2)$ and inserted tuple $(4, 4)$. v_2 updated tuple $(1, 1)$ to $(1, 42)$, deleted tuple $(2, 2)$, and inserted tuples $(3, 3)$ and $(4, 4)$.

<u>a</u>	b
1	1
2	2

(a) R

<u>a</u>	b
4	4

(b) R_1

<u>a</u>	b
1	42
3	3
4	4

(c) R_2

Figure 8: Initial database state R as well as states R_1 and R_2 produced by versions v_1 and v_2. Column a is underlined because it is a primary key.

The most straightforward way to show the diff of R_1 and R_2 is a side-by-side two-way tabular diff as shown in Figure 9a. The two-way diff is useful for quickly examining how R_1 and R_2 differ, but the two-way diff is missing some important context. Specifically, it is ambiguous how the differences in a two-way diff were produced.

Looking only at the two-way diff in Figure 9a, it is unclear whether v_1 deleted tuple $(1, 42)$ from R, v_2 added tuple $(1, 42)$ to R, or v_1 deleted some tuple with primary key 1 and then v_2 updated the tuple to $(1, 42)$. The two-way diff also does not show that both v_1 and v_2 deleted tuple $(2, 2)$ and added tuple $(4, 4)$.

R_1		R_2	
a	**b**	**a**	**b**
—	—	1	42
—	—	3	3

(a) A two-way diff of R_1 and R_2.

R_1		R		R_2	
a	**b**	**a**	**b**	**a**	**b**
—	—	1	1	1	42
—	—	2	2	—	—
—	—	—	—	3	3
4	4	—	—	4	4

(b) A three-way diff of R, R_1, and R_2.

Figure 9: Example two-way and three-way diffs of the relations in Figure 8.

To address the shortcomings of two-way diffs, we introduce three-way diffs, similar to the three-way diffs used by version control systems like git [7] when resolving merge conflicts. A three-way diff, like the one shown in Figure 9b, shows the difference between the original database state R and the two derived states R_1 and R_2.

The first row in the three-way diff in Figure 9b shows that the tuple $(1, 1)$ exists in the original state R, but was deleted in R_1 and modified in R_2. The second row shows that both v_1 and v_2 deleted tuple $(2, 2)$. The third row shows that v_2 added tuple $(3, 3)$. The fourth row shows that both v_1 and v_2 added tuple $(4, 4)$.

7.1.2 *Without Primary Keys.* Diffs of tables *without* primary keys are similar to, but also a bit noisier than, diffs of tables with primary keys. Consider the relation $R(a, b)$ without any primary key columns. States R, R_1, and R_2 are shown in Figure 10.

a	**b**
4	4
5	5
6	6

(a) R

a	**b**
1	1
2	2
6	6

(b) R_1

a	**b**
2	2
3	3
4	4

(c) R_2

Figure 10: Initial database state R as well as states R_1 and R_2 produced by versions v_1 and v_2. There are no primary keys.

The two-way diff of R_1 and R_2 is shown in Figure 11a. As with two-way diffs of tables with primary keys, the two-way diffs are helpful but ambiguous. Did v_1 insert tuple $(1, 1)$ or did v_2 delete tuple $(1, 1)$?

We similarly introduce three-way diffs, as shown in Figure 11b. Rather than grouping changes by primary key (remember there is no primary key), we group changes into one of six categories:

(1) Tuples that were inserted into R_1 but not R_2.
(2) Tuples that were inserted into R_1 and R_2.
(3) Tuples that were inserted into R_2 but not R_1.
(4) Tuples that were deleted from R_1 but not R_2.
(5) Tuples that were deleted from R_1 and R_2.
(6) Tuples that were deleted from R_2 but not R_1.

R_1		R_2	
a	**b**	**a**	**b**
1	1	—	—
6	6	—	—
—	—	3	3
—	—	4	4

(a) A two-way diff of R_1 and R_2.

R_1		R		R_2	
a	**b**	**a**	**b**	**a**	**b**
1	1	—	—	—	—
2	2	—	—	2	2
—	—	—	—	3	3
—	—	4	4	4	4
—	—	5	5	—	—
6	6	6	6	—	—

(b) A three-way diff of R, R_1, and R_2.

Figure 11: Example two-way and three-way diffs of the relations in Figure 10.

7.2 Computing Diffs

We now explain how to efficiently compute the three-way diff of relations R, R_1, and R_2 both with and without primary keys. Recall that R_1 and R_2 are branches and are implemented with relations R_1^+, R_1^-, R_2^+, and R_2^-. We compute the three-way diff in time proportional to the size of these "plus" and "minus" tables, not the size of the underlying table R. Thus, computing a diff is fast as long as the diff is small (the common case), regardless of how big the original data is. In contrast, the running time of a naive SQL-based approach to diffing relations is proportional to the size of the relations themselves. In Section 8, we show that our diff algorithm can be upwards of 100× faster than the naive approach.

7.2.1 *With Primary Keys.* We first look at how to compute the diff of two relations with primary keys.

Step 1. First, we prune duplicate rows that appear in both a plus table R_i^+ and minus table R_i^-. We use the letter P for "pruned".

$$P_i^+ = R_i^+ - R_i^- \qquad \text{for } i \in \{1, 2\}$$
$$P_i^- = R_i^- - R_i^+ \qquad \text{for } i \in \{1, 2\}$$

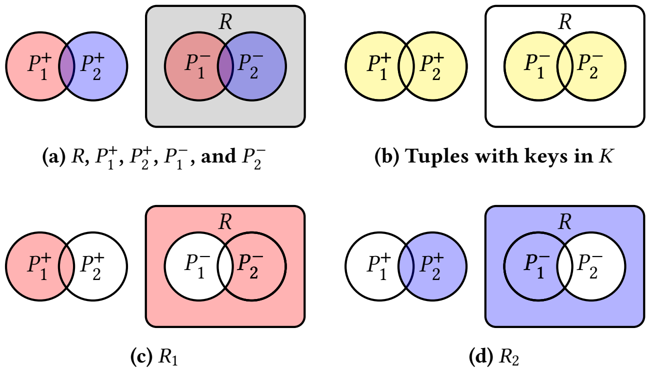
Step 2. Second, we collect the set of primary keys, K, for all rows modified by v_1 and v_2. Let π be a function that removes all but the primary keys columns of R.

$$K = \pi(P_1^+) \cup \pi(P_1^-) \cup \pi(P_2^+) \cup \pi(P_2^-)$$

Step 3. For every key k in K, we want to find the corresponding tuple (if any) in R, R_1, and R_2. We can accomplish this by taking the left outer joins $K ⟕ R$, $K ⟕ R_1$, and $K ⟕ R_2$, and then combining the three results, grouping by primary key. However, computing these left outer joins takes time proportional to the size of R. To avoid this, we rewrite the left outer joins to equivalent queries that only access P_1^+, P_1^-, P_2^+, and P_2^-.

$$K ⟕ R = K ⟕ (P_1^- \cup P_2^-)$$
$$K ⟕ R_1 = K ⟕ (P_1^+ \cup (P_2^- - P_1^-))$$
$$K ⟕ R_2 = K ⟕ (P_2^+ \cup (P_1^- - P_2^-))$$

These queries are explained pictorially in Figure 12. P_1^+ and P_2^+ are disjoint from R, and P_1^- and P_2^- are subsets of R, so they can drawn as the Venn diagram in Figure 12a. Figure 12b shows the tuples with primary keys in K. These are the tuples that will appear in the diff. Figure 12c and Figure 12d show the branched relations R_1 and R_2. Inspecting the diagrams, it's clear that the intersection of K and R is $P_1^- \cup P_2^-$; the intersection of K and R_1 is $P_1^+ \cup (P_2^- - P_1^-)$; and the intersection of K and R_2 is $P_2^+ \cup (P_1^- - P_2^-)$.



R
P_1^+
P_2^+
P_1^-
P_2^-
(a) R, P_1^+, P_2^+, P_1^-, and P_2^-
(b) Tuples with keys in K
(c) R_1
(d) R_2

Figure 12: An illustration of our three-way diff computation.

7.2.2 *Without Primary Keys.*

As explained in Section 7.1.2, we divide a diff without primary keys into six categories. We compute these categories explicitly:

(1) $P_1^+ - P_2^+$ (inserted into R_1 but not R_2)
(2) $P_1^+ \cap P_2^+$ (inserted into R_1 and R_2)
(3) $P_2^+ - P_1^+$ (inserted into R_2 but not R_1)
(4) $R_1^- - R_2^-$ (deleted from R_1 but not R_2)
(5) $R_1^- \cap R_2^-$ (deleted from R_1 and R_2)
(6) $R_2^- - R_1^-$ (deleted from R_2 but not R_1)

8 EVALUATION

We evaluated our diff testing framework on real-world bugs that led to production outages within Google, as described in Section 2. Our key findings are:

- Our framework was able to identify these bugs and provide meaningful diffs to the user.
- Our framework creates branches and compute diffs twice as fast as implementations built using Postgres [22] or Dolt [6], and introduces modest overhead to reads and writes, making our end-to-end diff testing both practical and capable of providing timely diff results to the user.

In the following, the graphs show table rows or database rows on a logarithmic scale.

8.1 Setup

Implementation. We implemented our diff testing framework in Go [8]. The framework takes an initial database state, binaries v_1 and v_2, and a request log. It then repeatedly executes the workflow, performing the branching, replaying, and diffing stages for a number of request interleavings.

Workloads. We recreated each bug category from Section 2 in a widely used open-source application called Bank of Anthos [1] which demonstrates how to implement a bank using modern development practices.

Baselines. We compared the performance of our framework against Postgres [22] and a versioned database Dolt [6]. Postgres does not support branching or diffing, but we use it to establish a performance baseline. Dolt is an open-source versioned database service that supports built-in branching and diffing.

8.2 Finding Bugs

In Section 2, we identified three categories of bugs that occur in practice in stateful applications that are challenging to find using existing testing techniques. We recreated these bugs in Bank of Anthos [1] and tested the buggy version of the application using our diff testing framework.

To set up a diff testing for Bank of Anthos [1], it took an engineer three days to create a testing environment for the application. Then it took an engineer one day to generate valid requests for diff testing.

8.2.1 *Data Corruption.*

We reproduced this bug by writing corrupt data into the database for a small fraction of queries. The buggy version of the application ran successfully without crashing or returning any unexpected client responses. However, our diff testing framework successfully flagged the corrupted data values when diffing database states.

8.2.2 *Data Incompatibility.*

We reproduced this bug by introducing a string-valued column to one of application's databases. The original version v_1 and the buggy version v_2 of the application differ in how long they expect the string to be. As in the production incident, v_1 and v_2 work as expected in isolation, and v_2 is backward-compatible with v_1. However, v_1 is not forward-compatible with v_2, leading to the production outage. Our diff testing framework was able to find the bug by interleaving requests between v_1 and v_2.

8.2.3 *False Data Assumptions.*

We reproduced this bug by setting a hard-coded upper limit on the expected size of a table in a database. In the buggy version v_2, the hard-coded upper limit is set to be smaller than the size of the production table. This causes v_2 to ignore some rows in the table, which leads to v_2 incorrectly processing some requests. This bug is not caught when testing against a small database. Since our diff testing framework supports lightweight branching, we were able to run diff tests against a copy of the entire production database to catch the bug, even though the production database is very large.

8.3 Microbenchmarks

We benchmarked the performance of four major branch related operations, as shown in Table 1: creating a branch, reading from a branch, writing to a branch, and diffing two branches. The performance of these operations is crucial in determining the feasibility of any stateful testing solution in practice.

The operations were benchmarked on both primary key tables and non-primary key tables. This split was necessary due to the different branching implementations for the two types of tables (Section 5), which exhibit different performance characteristics.

All the benchmarks were performed against tables that have roughly 20,000, 100,000, 600,000, 1,000,000 and 2,000,000 rows, where each row has two fields, and each field is between 5 and 20 bytes. We create two tables in the same database for the benchmarks, as shown in Figure 13. The two tables have a similar schema; the

Table 1: A summary of our microbenchmarks. We compare our framework against Dolt and Postgres.

	R^+/R^-	Dolt	Postgres
Branch	Create a branch	Create a branch	Copy database
Read	Read from branch	Read from branch	Read from table
Write	Write to branch	Write to branch	Write to table
Diff	3-way branch diff	2-way branch diff	Naive SQL queries

only difference is that users_pk has a primary key column, while users does not. The databases are populated with random data. All benchmarks assume that the size of the database is much larger than the size of the changes made to the database, which is the common case.

```
CREATE TABLE users(
    username VARCHAR(64) NOT NULL,
    password VARCHAR(64) NOT NULL);

CREATE TABLE users_pk(
    username VARCHAR(64) PRIMARY KEY,
    password VARCHAR(64) NOT NULL);
```

Figure 13: The two tables used throughout our benchmarks

8.3.1 Branching. In Figure 14, we show the time it takes to create a branch as a function of the size of the database (two tables included). Postgres does not support branching, so we show the time it takes to copy the database in its entirety. Note that the graph shows the latency (the y-axis) on a logarithmic scale.

Dolt requires roughly 10 milliseconds to create a branch regardless of the size of the database. Our framework requires roughly 100 milliseconds, also regardless of the size of the database. To create a branch of relation R, our framework creates auxiliary relations R^+, R^-, and R' and registers a number of triggers. All these operations are independent of the size of the database. The time to copy a Postgres database expectedly increases with the size of the database and is between 10× and 100× slower than our framework for the database sizes we evaluated.

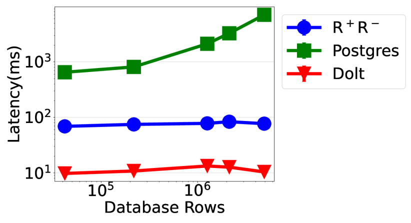


Figure 14: Branch latency for entire database

8.3.2 Read Performance. To measure the performance of reads performed against a branch, we execute the queries shown in Figure 15. We picked these queries because they have different query execution plans and appear commonly in practice. For Postgres, we execute the queries against regular tables, as there is no branching.

We run each query 250 times. The average latency across these runs is shown in Figures 16, 17, and 18. The standard deviations of the measurements are shown as error bars in the graphs.

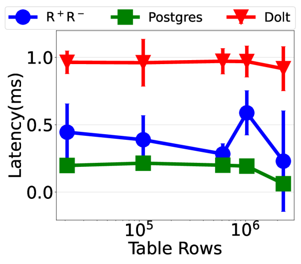
For Figure 16, we execute a point query to get either 0 or 1 records for the table with primary key, and ~ 10 records for the table without primary key. For Figure 17, we ran a short-range scan to get ~0.5% of all rows. For Figure 18, we scan the full table.

```
-- Query 1: Retrieve a small set of records based on
-- a single, exact match on a specific attribute.
SELECT * FROM users
WHERE username = 'aaaaaaaaaa';

-- Query 2: Retrieve a small set of records based on
-- conditional filters and pattern matches.
SELECT * FROM users
WHERE LENGTH(password) = 8 AND username LIKE 'a%%';

-- Query 3: A full table scan.
SELECT * FROM users;
```

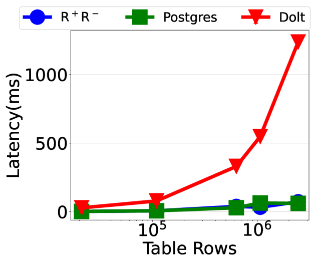
Figure 15: The queries we run against the users table. The queries for the users_pk table are identical.

From Figures 16 and 17, we see that our prototype introduces a very modest read overhead compared to Postgres, with 3× higher latency in the worst case. Dolt latency is on par with our framework for tables with primary keys, but is significantly worse for tables without primary keys.

Figure 18 shows that our solution does introduce significant read overheads for full table scans of tables without primary keys and slight but non-negligible read overheads for tables with primary keys. However, full table scans of large tables without primary keys are rare in practice and should be avoided, so we left this optimization as future work.



(a) With Primary Key

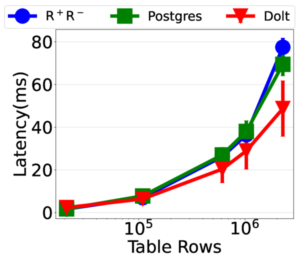


(b) Without Primary Key

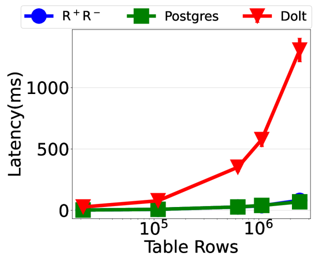
Figure 16: Query 1 Latency

8.3.3 Write Performance. Figure 19 shows the latency of inserting a row into a branch as a function of the size of the database. For Postgres, the rows are inserted into a regular table. We perform the insertion 1,000 times and plot the average write latency.

Our framework introduces a negligible amount of write overhead compared to Postgres. The overhead comes from the trigger that is executed whenever a row is inserted into a branch. This trigger

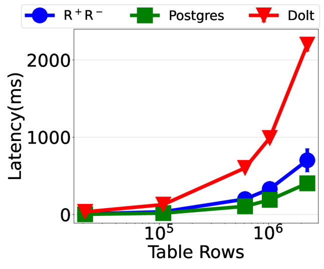


(a) With Primary Key

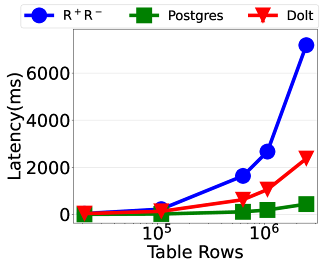


(b) Without Primary Key

Figure 17: Query 2 Latency



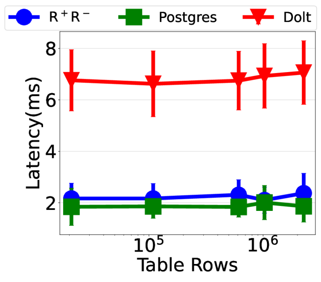
(a) With Primary Key



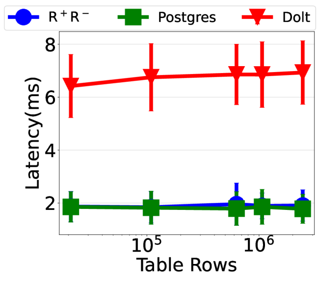
(b) Without Primary Key

Figure 18: Query 3 Latency

validates database constraints and forwards the writes to R^+ or R^-. These overheads are constant, regardless of the size of the database. Dolt is roughly 3× to 4× slower than our framework and Postgres.



(a) With Primary Key



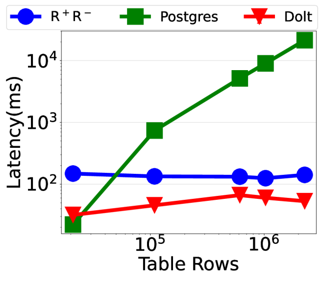
(b) Without Primary Key

Figure 19: Write Latency

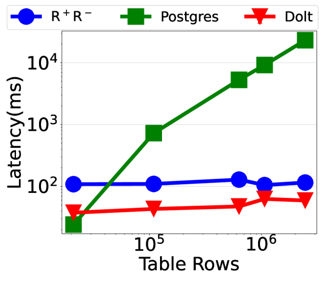
8.3.4 Diffing. Figure 20 shows the latency of computing the diff of two branches as a function of the size of the database. Note that the latency (y-axis) is shown on a logarithmic scale. Postgres does not support built-in diffing, so we instead run the following queries to compute a naive two-way diff between relations A and B:

```
SELECT * FROM A EXCEPT ALL SELECT * FROM B;
SELECT * FROM B EXCEPT ALL SELECT * FROM A;
```

Our framework computes diffs in approximately 100 milliseconds, which is roughly 2× to 3× slower than Dolt. Also note that the time to compute a diff does not increase with the size of the database. As explained in Section 7, the time to compute a diff is determined by the size of the diff rather than the size of the database. In practice, the sizes of the diffs are significantly smaller than the sizes of the database. In contrast, the naive approach to computing diffs using Postgres takes time proportional to the size of the database and is up to 1,000× slower than our framework.



(a) With Primary Key



(b) Without Primary Key

Figure 20: Diff Latency

8.3.5 Interleaving. There are 2^n total ways to interleave n requests between two versions, but most bugs require far fewer interleavings to manifest. Data corruption and false data assumption bugs often don't require interleaving at all. Data incompatibility bugs often involve two requests interleaved across the two versions, which requires only four interleavings in expectation.

For example, consider a sequence of n requests where two requests r_i and r_j are susceptible to a forward compatibility bug. That is, if r_i is run in v_2 and r_j is run on v_1, then a bug is triggered. In expectation, it requires only four random interleavings of the request sequence to find such a bug.

In general, it is possible that some pathological bugs might only manifest given a peculiar interleaving of requests. If a bug requires a specific interleaving of m requests in a sequence of n total requests, then it will take 2^m interleavings to find the bug in expectation. However, we believe m is small for the majority of bugs.

8.3.6 End-to-End Runtime. Since branching and diffing introduce almost no overhead, and read and write operations introduce modest overhead, the end-to-end runtime of our framework is mainly determined by the time it takes to execute a request against the application being tested, the number of requests to be executed, and the number of interleavings that are explored.

Table 2 shows the time and number of interleaving our framework required to find the bugs outlined in Section 8.2. For comparison, executing the same requests on an unmodified Postgres database takes 27.06 seconds.

Table 2: Time required to find the bugs in Section 8.2

	Time	Interleavings
Data Corruption Bug (Sec 8.2.1)	55.88 s	2
Data Incompatibility Bug (Sec 8.2.2)	110.48 s	4
False Data Assumption Bug (Sec 8.2.3)	55.31 s	2

8.4 Sensitivity analysis

To accurately assess our solution in a practical environment, we repeat the branching, diffing, and read/write benchmarks in Section 8.3 but with larger database sizes. We benchmark our framework on tables that have roughly 20,000, 600,000, 2,000,000, 11,000,000, 40,000,000 rows. Each database has one table with a primary key and one table without a primary key.

Figure 21 shows that **branch** latency is independent of database size. Figure 22, 23, and 24 show that **read** latency increases with database size for the three queries in Figure 15. This is expected because, as the size of the database increases, the queries need to scan more rows and pages to find the required data. Figure 25 shows that **writes** continue to introduce negligible overhead when the database size grows. Figure 26 shows **diff** latency for various numbers of modified rows. Diff latency increases as the number of modified rows increase, but it is independent of database size.

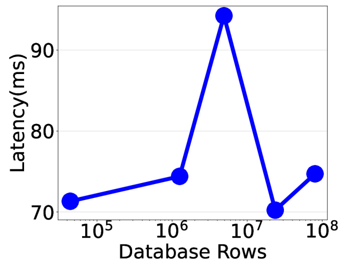
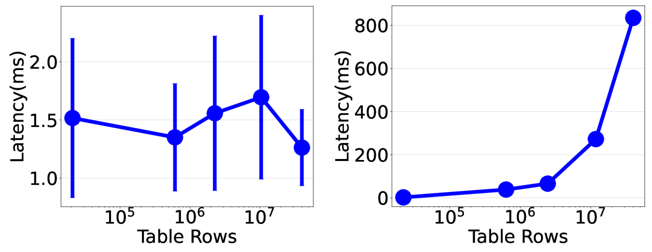
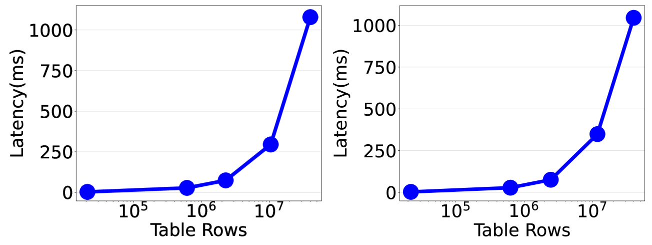


Figure 21: Branch latency for entire database



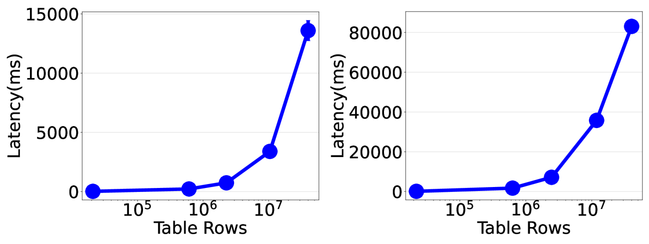
(a) With Primary Key **(b) Without Primary Key**

Figure 22: Query 1 Latency



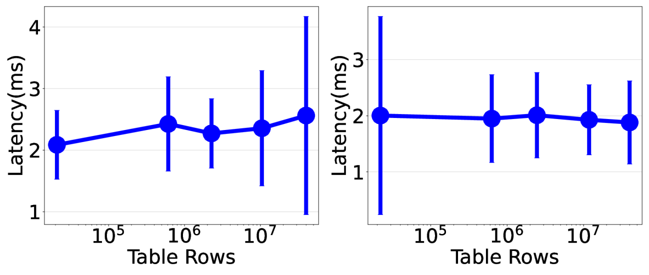
(a) With Primary Key **(b) Without Primary Key**

Figure 23: Query 2 Latency



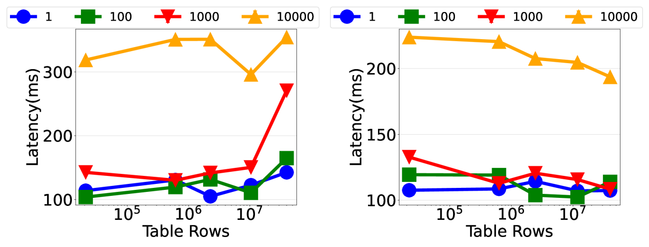
(a) With Primary Key **(b) Without Primary Key**

Figure 24: Query 3 Latency



(a) With Primary Key **(b) Without Primary Key**

Figure 25: Write Latency



(a) With Primary Key **(b) Without Primary Key**

Figure 26: Diff Latency

8.5 Macrobenchmark with YCSB

Yahoo! Cloud Serving Benchmark (YCSB) [4] is an open-source specification and program suite to simulate real-world workloads for benchmarking database management systems. We benchmark the read and write performance of our approach with all six core workloads [31] of YCSB. Workloads A-F are write-heavy, read-heavy, read-only, read latest, short range reads, and read-modify-write workloads respectively. We load 50,000,000 rows for each workload, and each workload used 1,000-byte records with ten 100-byte fields each. During the benchmarks, we execute 1000 operations per workload with uniform request distribution. Because our framework currently does not support concurrency, we set the number of threads to be one. Figure 27 shows the benchmark results compared against vanilla Postgres. Our solution introduces modest overhead for all workloads. The standard deviations of the measurements are shown as error bars in the graphs.

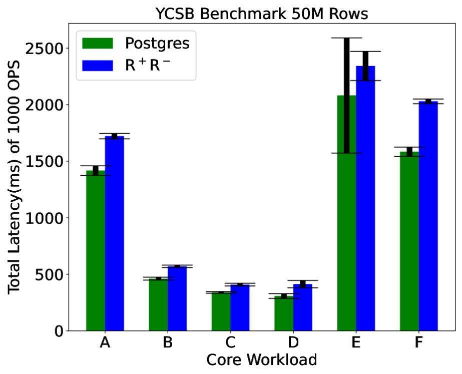


Figure 27: YCSB benchmark results.

9 RELATED WORK

Versioned Databases. Across academia and industry, there are many versioned databases [6, 11, 14, 16, 27]. For instance, OrpheusDB [11] introduces a dataset version control system that is "bolted-on" to a relational database. Decibel [14] describes a new version-oriented storage engine designed "from the ground up" to support versioning. These databases target collaborative data science while we target stateful diff testing. This difference in focus led to very different software implementations.

Take OrpheusDB as an example, which has poor performance on diff testing workloads. Given a relation $R(a, b)$, OrpheusDB create two tables: one for data and one for version metadata. The data table is a copy of R with a special *rid* column prepended. The versioning table contains one row for every version, and the row contains a list of all the *rid*s in that version.

Creating a branch of a versioned OrpheusDB table is a two-step procedure. First, you must export the versioned table into a CSV file or into a plain Postgres table. This requires a full copy of the entire database. After you make modifications to the exported table, you can import the table back into OrpheusDB, which again does work proportional to the size of the database.

OrpheusDB does not support in-place writes. To modify a row of a table, you must export the table, modify the row externally to OrpheusDB, and then import the modified table back into OrpheusDB. Thus, every write requires a full database copy.

To understand how OrpheusDB executes reads, consider the OrpheusDB query `SELECT COUNT(*) FROM VERSION 1 OF CVD R` which returns the number of rows in version 1 of relation R. OrpheusDB implements this query by first fetching the *rid*s for version 1 from the version metadata table (e.g., $\{1, 2, 3\}$). It concatenates these versions into a string and then injects them into a query that looks like the following:

```
SELECT COUNT(*)
FROM Rdata
WHERE rid = ANY('{1, 2, 3}'::int[])
```

The time to *construct* the query requires time proportional to the size of the table. The query itself is slow because of the additional `WHERE rid = ANY(...)` clause. The *size* of the query is proportional to the size of the database.

Empirically, running `SELECT COUNT(*) FROM R` using plain Postgres on a relation with one million rows takes 0.855 milliseconds. Running the query using OrpheusDB is over **2,500× slower**, taking 2.26 seconds. This behavior might be acceptable for collaborative data science workloads where relations are small and modifications infrequent. But for stateful diff testing, where relations are huge and modifications are very frequent, OrpheusDB unacceptably slow.

Dolt [6] and Neon [16] are mature versioned database products used by several companies. Neon uses copy-on-write techniques and supports branching but not diffing. Dolt uses a Prolly [24] tree data structure to implement the storage layer and supports both branching and diffing. These databases are implemented as separate database services. It is not practical to integrate them into a production testing framework, since it would require developers to connect to a different database service, migrate all their production data, and change the queries in their code.

Testing Stateful Services. In [3], Chays *et al.* introduce a framework for testing database applications. They introduce a tool for populating the database with meaningful data that satisfies database constraints. However, this data is not guaranteed to mimic the production environment. For example, some bugs only occur with certain database sizes. In addition, developers have to replicate the database for each individual test.

Detecting Rollout Failures. In [32], Zhang *et al.* studied the root cause of upgrade failures and concluded that data syntax and semantics incompatibility are the main reasons for incompatible cross-version interactions. They simulate a three node cluster of the target distributed system to test full-stop upgrades, rolling upgrades, and new nodes joining the system.

Mvedsua [21] introduced a novel approach to dynamic software updating, using multi-version execution to update in-memory, stateful applications. Mvedsua mainly focuses on updating live, in-memory stateful service, while our work concentrates on diff testing stateful services backed by a relational database.

Database Snapshots. Most cloud relational databases support point-in-time recovery [29] by using write-ahead logging [30]. Using point-in-time recovery, we can restore a database at a given timestamp. Only some databases (*e.g.*, Oracle [19]) support flashback `AS OF` queries, which return database objects as of a previous point in time. We recommend developers use database snapshots to create the initial database state to test against. However, on its own, database snapshots are too inefficient to be used for diff testing. Efficient branching is needed in conjunction with snapshots.

10 CONCLUSION

In this paper, we presented a practical stateful diff testing framework that leverages a combination of three techniques. First, our "bolt-on" approach to branching databases allows the framework to test against entire production databases without having to make expensive database copies. Second, our framework computes three-way relational diffs between database branches in time proportional to the size of the diffs. Third, our framework interleaves request sequences between software versions to increase test coverage. We showed that our framework is able to identify common categories of catastrophic bugs that occur in practice during the rollout of stateful applications with minimal performance overhead.

REFERENCES

[1] Bank of Anthos [n.d.]. BankOfAnthos. https://github.com/GoogleCloudPlatform/bank-of-anthos.

[2] Canarying [n.d.]. Canarying Releases. https://sre.google/workbook/canarying-releases/.

[3] David Chays, Saikat Dan, Phyllis G Frankl, Filippos I Vokolos, and Elaine J Weyuker. 2000. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. 147–157.

[4] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[5] diffy 2024. Diffy. https://github.com/opendiffy/diffy.

[6] Dolt [n.d.]. Dolt. https://docs.dolthub.com/.

[7] Git [n.d.]. Git. https://git-scm.com/.

[8] Go [n.d.]. The Go programming language. https://go.dev/.

[9] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.

[10] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 71–80.

[11] Silu Huang, Liqi Xu, Jialin Liu, Aaron Elmore, and Aditya Parameswaran. 2017. Orpheusdb: Bolt-on versioning for relational databases. *arXiv preprint arXiv:1703.02475* (2017).

[12] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[13] James Wayne Hunt and M Douglas MacIlroy. 1976. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill.

[14] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 9. NIH Public Access, 624.

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

[16] neon [n.d.]. Neon Serverless Postgres. https://neon.tech/.

[17] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid differential software analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1273–1285.

[18] Oracle [n.d.]. Oracle Database. https://www.oracle.com/database/.

[19] Oracle Flashback Query [n.d.]. Oracle Flashback Query. https://docs.oracle.com/cd/B14117_01/appdev.101/b10795/adfns_fl.htm.

[20] Oracle Server [n.d.]. SQL Server. https://www.microsoft.com/en-us/sql-server.

[21] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 573–585.

[22] Postgres [n.d.]. PostgreSQL. https://www.postgresql.org/.

[23] Postgres Trigger on Views [n.d.]. PostgreSQL Trigger On View. https://www.postgresql.org/docs/current/trigger-definition.

[24] Prolly Trees [n.d.]. Prolly Trees. https://docs.dolthub.com/architecture/storage-engine/prolly-tree.

[25] Protobuf [n.d.]. Protocol Buffers. https://protobuf.dev/.

[26] Quora. [n.d.]. What is meant by a rolling upgrade in software development. https://www.quora.com/What-is-meant-by-a-rolling-upgrade-in-software-development.

[27] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Wanzeng Fu, Beng Chin Ooi, and Pingcheng Ruan. 2018. Forkbase: An efficient storage engine for blockchain and forkable applications. *arXiv preprint arXiv:1802.04949* (2018).

[28] Wikipedia. 2024. Differential testing — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Differential%20testing&oldid=1186195819.

[29] Wikipedia. 2024. Point-in-time recovery — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Point-in-time_recovery.

[30] Wikipedia. 2024. Write-ahead logging — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Write-ahead_logging.

[31] ycsb 2020. YCSB Benchmark Core Workloads Wiki. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[32] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and detecting software upgrade failures in distributed systems. In *SOSP*.