Transactions Make Debugging Easy

Qian Li, **Peter Kraft**, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia

Qian Did All The Work

A Real Bug in moodle

- A popular online education platform.
- Users can subscribe to a forum and create posts in it.
- All persistent state is managed in Postgres.

A Real Bug in moodle

- MDL-59854: Users can get registered for the same forum twice.
- Developers struggled to reproduce the bug and took 3 months to release the bug fix.

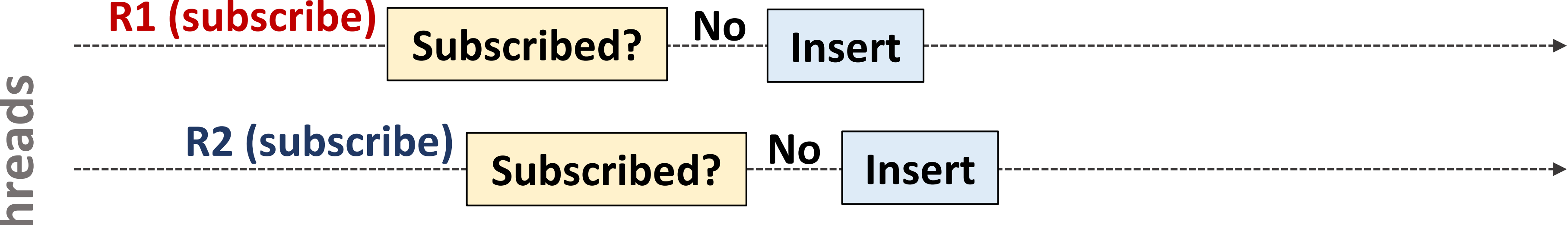
> Since the time window, between the execution of the two line, responsible for the race condition, is pretty small. **You have to be pretty fast and pretty lucky to actually reproduce this issue** (we have only 14 occurrences in 190111 forum subscriptions).

A Real Bug in moodle

- If a user hasn't subscribed to a forum, insert a subscription to the database table.
- `isSubscribed` and `forumInsert` run in separate transactions.

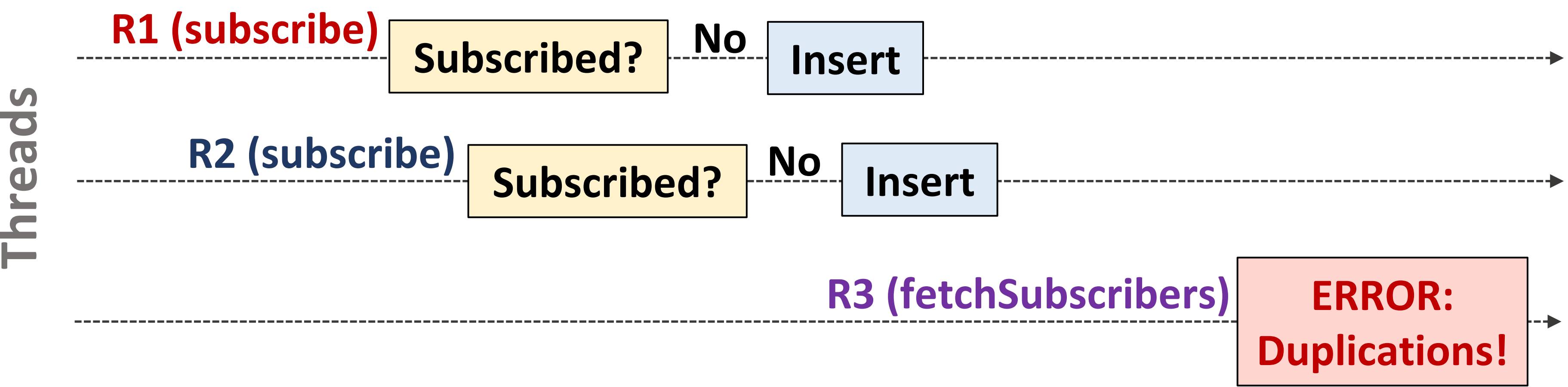
```
1 def subscribeUser(userId, forum):
2   if (not isSubscribed(userId, forum)):
3     forumInsert(userId, forum)
```

A Classic Time-of-Check/Time-of-Use Bug



- R2 started before R1's insert and didn't see the change. Both requests succeeded.

A Classic Time-of-Check/Time-of-Use Bug



- R2 started before R1's insert and didn't see the change. Both requests succeeded.
- A later request failed when it fetched the subscribers.

Existing Tools Are Not Enough

```
Did you remember to make the first column something unique in your call
to get_records? Duplicate value '24028' found in column 'id'.

  • line 807 of /lib/dml/pgsql_native_moodle_database.php: call to
    debugging()
  • line 456 of /mod/forum/classes/subscriptions.php: call to
    pgsql_native_moodle_database->get_records_sql()
  • line 114 of /mod/forum/subscribers.php: call to
    mod_forum\subscriptions::fetch_subscribed_users()
```

Conventional error messages and stack traces only provide information for the failed request, but not the root cause.

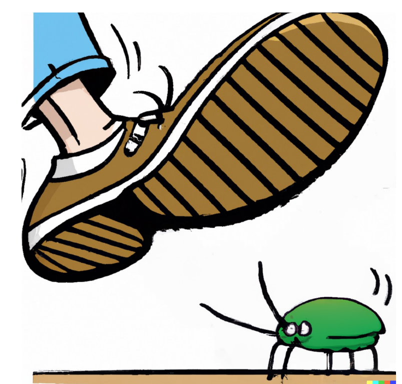
How Can We Simplify Debugging?

Recent trends in cloud applications:

- Modern applications have little local persistent state and manage state in remote data stores.
    - E.g., Moodle, microservices and serverless applications.

- Many data stores are providing strong transactional semantics.

A Vision for TROD:
Transaction-Oriented Debugging



TROD Leverages Trends in Cloud Applications

TROD targets applications that follow three design principles:

1) Store all application shared state in databases.
2) Manage shared state only through ACID transactions.
3) Deterministic.

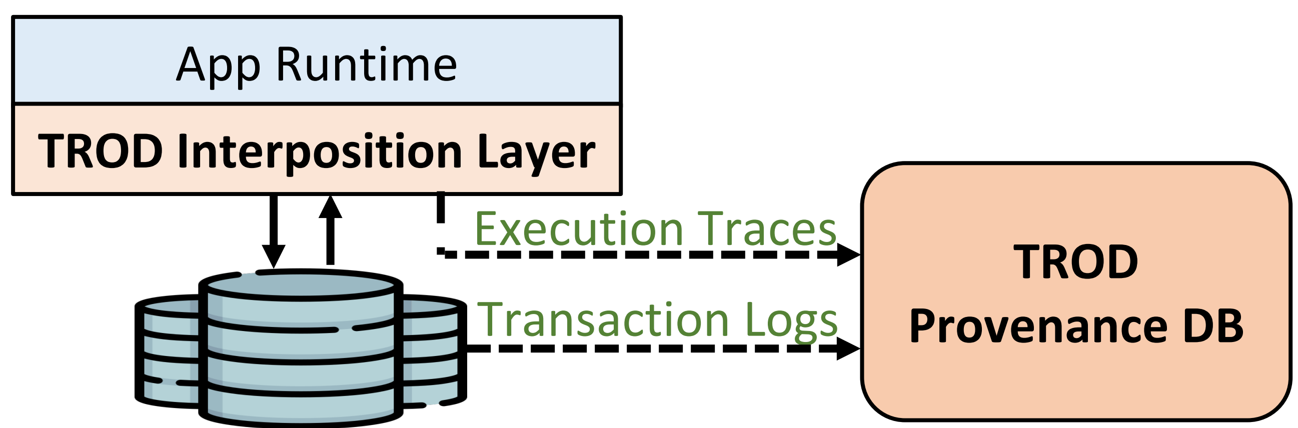
They align with recent trends in cloud applications.

TROD Principles Enable Powerful Features

- Automatic low-overhead tracing and declarative debugging.
- Faithful bug replay.
- Retroactive programming.

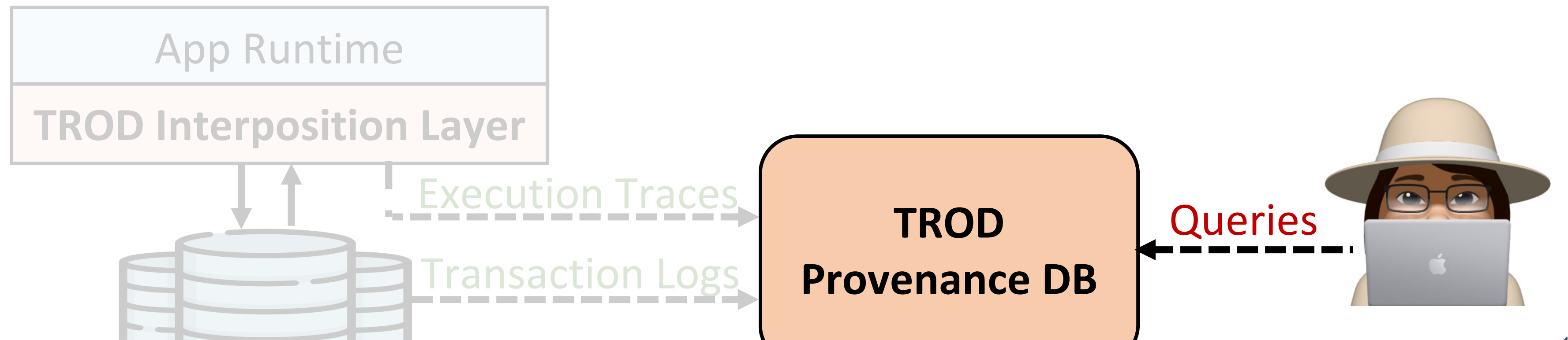
Automatic Tracing and Declarative Debugging

- TROD can intercept DB queries and automatically track app execution and data operations in a provenance store.
  - Drop-in replacement for data access libraries (e.g., JDBC).
  - Low overhead: Leverage Change-Data Capture and transaction logs.



Automatic Tracing and Declarative Debugging

- TROD can intercept DB queries and automatically track app execution and data operations in a provenance store.
- Developers can use a declarative language (SQL) to query this provenance store. E.g., to locate the root cause of a bug.



TROD Tracing Can Locate the Moodle Bug

- Change data log: What data items are written by each transaction. E.g., "*Find transactions that inserted (U1, F2)*"

Txn_ID	Query	UserId	Forum
1	Check if (U1, F2) exists	null	null
2	Check if (U1, F2) exists	null	null
3	Insert	U1	F2
4	Insert	U1	F2
...	...	...	...

Duplicated inserts!

TROD Tracing Can Locate the Moodle Bug

- TROD log: Transaction order and corresponding code.
  E.g., *"Check all executions that are relevant to duplicated (U1, F2)."*

Txn_ID	Timestamp	Exec_ID	Metadata
1	TS1	**R1**	**subscribeUser**:`isSubscribed`
2	TS2	**R2**	**subscribeUser**:`isSubscribed`
3	TS3	**R2**	**subscribeUser**:`forumInsert`
4	TS4	**R1**	**subscribeUser**:`forumInsert`
...	...	...	...

Root cause: interleaved subscribeUser executions

Faithful Replay

If apps are deterministic, and access shared state only transactionally, we can faithfully replay any past trace:

1. Re-execute code normally but stop before each transaction;
2. Restore the DB to an equivalent original state;
3. Re-execute the transaction.

Follow the transaction order obtained from the DBMS.

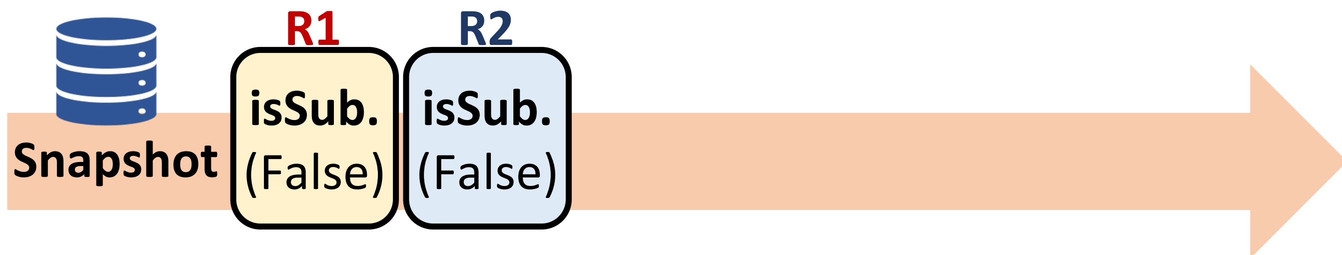
TROD Replay Can Reproduce the Moodle Bug

- Restore the DB to a snapshot right before subcribeUser executions.

TROD Replay Can Reproduce the Moodle Bug

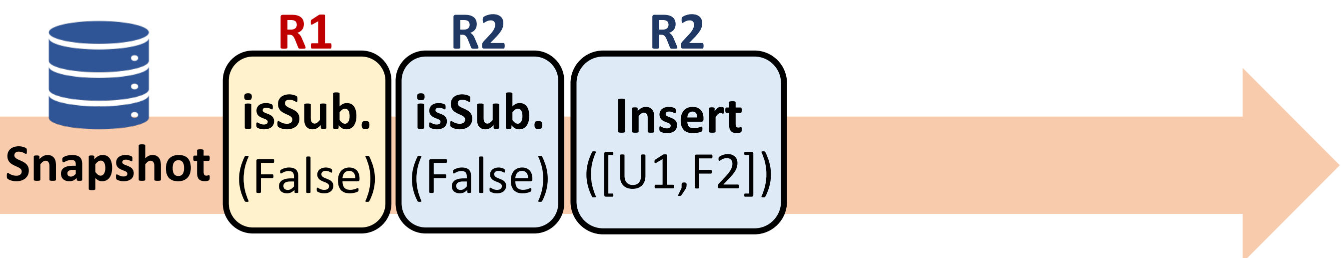
- Replay relevant transactions according to the execution log.

TROD Replay Can Reproduce the Moodle Bug



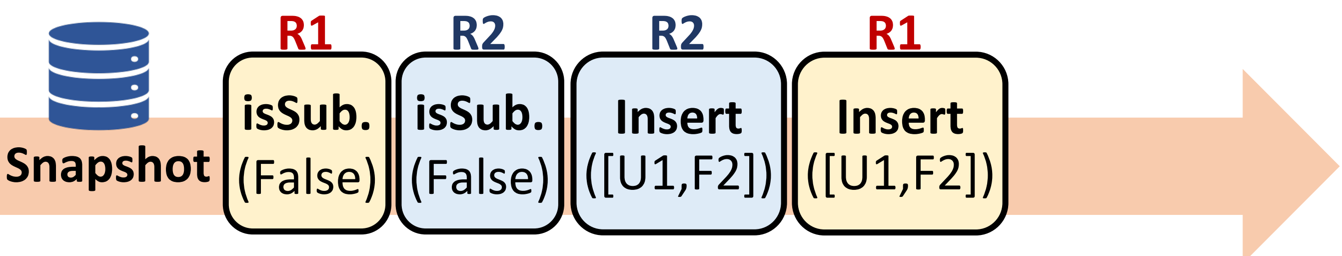
- Replay relevant transactions according to the execution log.

TROD Replay Can Reproduce the Moodle Bug



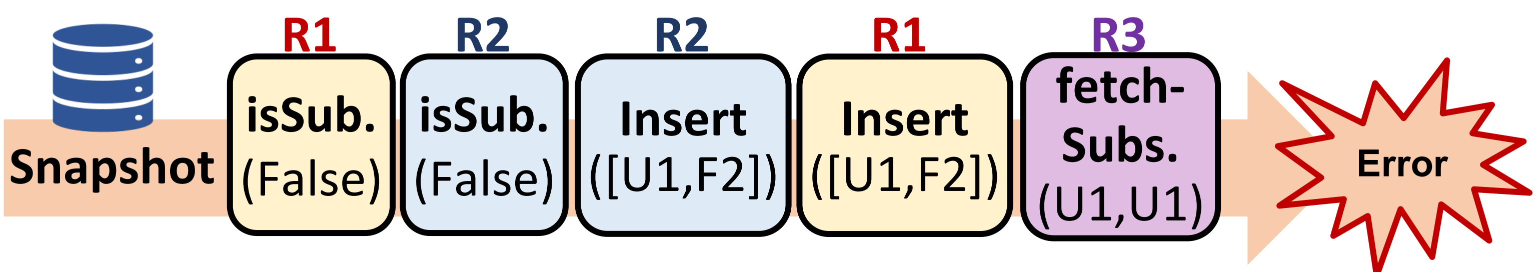
- Replay relevant transactions according to the execution log.
- Both R1 and R2 see no subscriptions and insert the same entry.

TROD Replay Can Reproduce the Moodle Bug



- Replay relevant transactions according to the execution log.
- Both R1 and R2 see no subscriptions and insert the same entry.

TROD Replay Can Reproduce the Moodle Bug



- Replay relevant transactions according to the execution log.
- Both R1 and R2 see no subscriptions and insert the same entry.
- The last request prints the same error message.

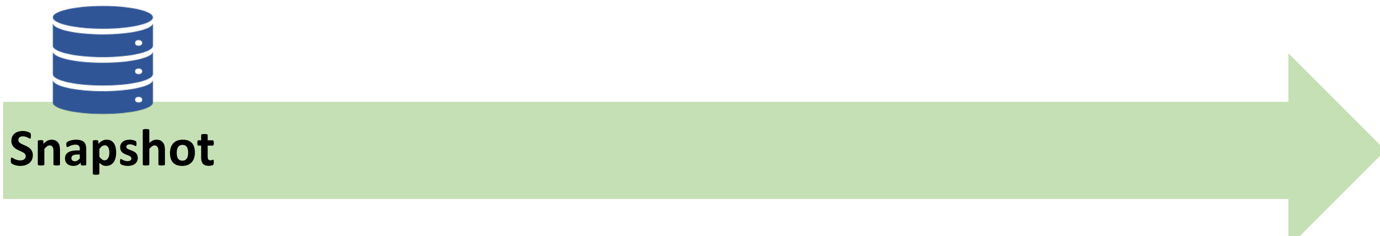
Retroactive Programming

- Based on faithful replay, TROD allows developers to **modify** their code and test it on past events: ***retroactive programming*.**
- Useful for testing whether a bug fix works and doesn't cause new bugs.

TROD Makes Retroactive Programming Practical

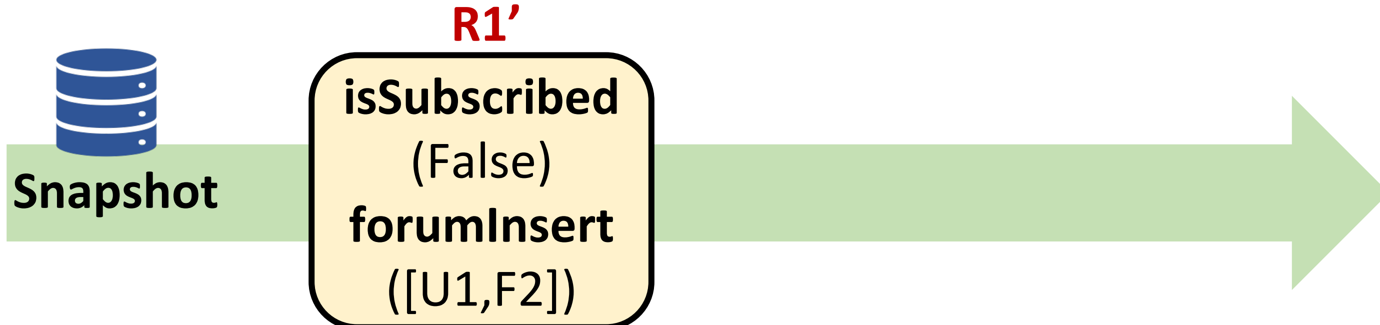
- Retroactive programming is challenging because it requires tracking causality.
  - Otherwise, we have to rerun everything even for a small change.
  - Infeasible to track every variable and memory address.
- But feasible in TROD because state is only accessed through transactions.
- We can track causality through data provenance and selectively re-execute traces.

TROD Retroactive Programming Can Test the Bug Fix



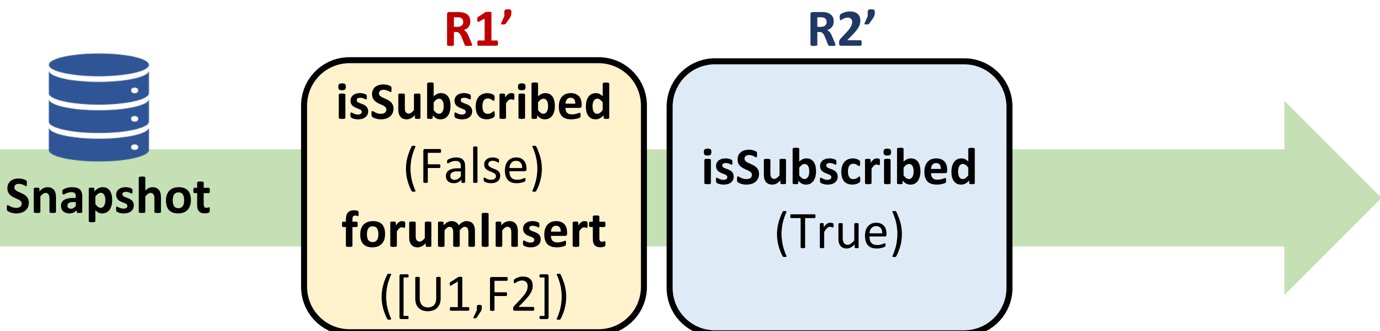
- Restore the DB to a snapshot right before subcribeUser executions.

TROD Retroactive Programming Can Test the Bug Fix



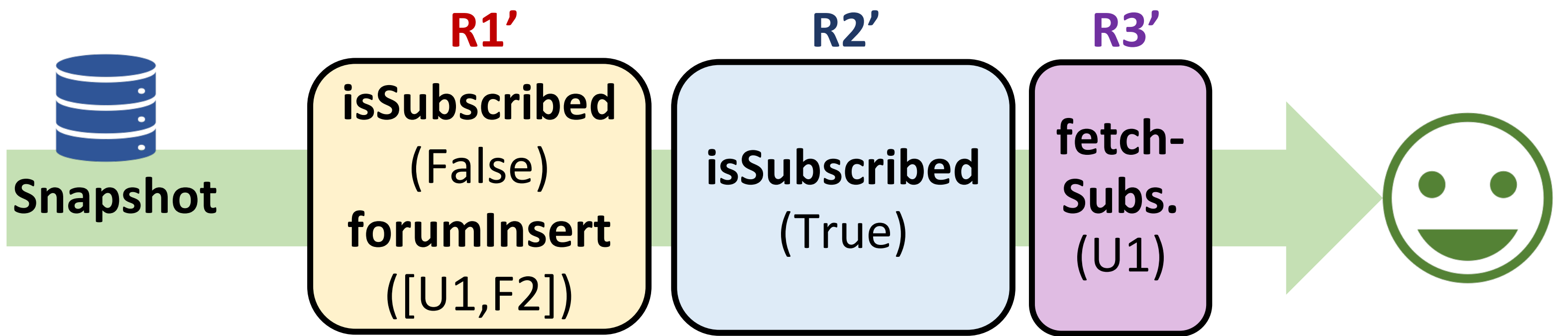
- Re-execute requests with the new code.

TROD Retroactive Programming Can Test the Bug Fix



- Re-execute requests with the new code.
- The second request no longer inserts a duplicated subscription.
  - Because *isSubScribed* and *Insert* run in one transaction, two concurrent requests cannot interleave.

TROD Retroactive Programming Can Test the Bug Fix



- Re-execute requests with the new code.
- The second request no longer inserts a duplicated subscription.
- The last request prints no more error messages – the bug fix works!

TROD in Action

```
qianli@qian-node-qjk8:~/trod$ java -jar target/demo-fat-exec.jar -mode exec -numReq 3
[WARN ] RetroDemo - Non-replay mode.
[ERROR] MDLFetchSubscribers - Duplicated subscriptions for forum 22, userId 11
qianli@qian-node-qjk8:~/trod$
qianli@qian-node-qjk8:~/trod$
```

[0] 0:bash* 1:docker 2:vim- "qian-node-qjk8" 21:32 04-Jan-23

Our Next Step

- How to restore the production database efficiently in a development environment?
- What do we replay and what can we skip?
  - Track dependencies through data provenance.
- For retroactive programming, in what order should concurrent requests execute?
  - Reduce the number of enumerations through dependency analysis.

Conclusion

- We present our vision for TROD: transaction-oriented debugging.
- We are actively developing and improving our prototype.
- The next time you have a bug, TROD on it!

DBOS project: https://dbos-project.github.io

Contact Qian: https://cs.stanford.edu/~qianli/

Looking forward to your feedback!