Report from Dagstuhl Seminar 23441

Ensuring the Reliability and Robustness of Database Management Systems

Hannes Mühleisen*[1], Danica Porobic*[2], and Manuel Rigger*[3]

1 **CWI – Amsterdam, NL.** hannes.muehleisen@cwi.nl
2 **Oracle Switzerland – Zürich, CH.** danica.porobic@oracle.com
3 **National University of Singapore, SG.** rigger@nus.edu.sg

Abstract

The goal of this Dagstuhl Seminar was to bring together researchers and practitioners interested and experienced in database systems and the reliability aspects thereof. It is a continuation of a previous seminar on this topic, which had built an initial understanding of the challenges and areas of future work. In this edition of the seminar, we aimed for a tangible outcome of the seminar by writing a manuscript documenting the (1) best practices in ensuring database systems' reliability, (2) the state of the art on this topic in research, as well as (3) open challenges, which might also serve as the cornerstone of writing a book on this topic presented in this report. We achieved this by forming four primary working groups during the seminar, namely (1) on the automated testing aspects concerning analytical components of database systems, (2) benchmarking, (3) reliability for transaction and concurrency aspects, as well as (4) query languages and debugging. The report contains four sections presenting the results of these working groups. Some of these working groups and individuals plan to further refine their work and discussion outcomes, aiming to submit them to upcoming venues.

Seminar October 29 – November 3, 2023 – https://www.dagstuhl.de/23441
2012 ACM Subject Classification Information systems → Database management system engines; Software and its engineering → Software verification and validation
Keywords and phrases database benchmarking, database reliability, database testing
Digital Object Identifier 10.4230/DagRep.13.10.139

1 Executive Summary

Manuel Rigger (National University of Singapore, SG)
Hannes Mühleisen (CWI – Amsterdam, NL)
Danica Porobic (Oracle Switzerland – Zürich, CH)

Database systems are an essential component of most software systems. It is crucial that they function correctly and are efficient. Achieving this is difficult, given that growing demands require increasingly sophisticated systems and adapting them to new hardware platforms. Building on the success of the last seminar, the goal of this Dagstuhl Seminar was to advance database systems reliability by bringing together both practitioners as well as researchers working in this domain. We discussed current practices, approaches, and open challenges in manual and automated correctness and performance testing, isolation-level testing, benchmarking, database and query generation, query languages, as well as debugging.

* Editor / Organizer

Goals and Outcomes

As a concrete tangible outcome of the seminar, we aimed to write a manuscript on (1) best practices in ensuring database systems' reliability, (2) the state of the art on this topic in research, as well as (3) open challenges, which might also serve as the cornerstone of writing a book on this topic. We believe we have achieved this goal, with the reports of the four working groups presented in the subsequent chapters. The majority of the contents were composed during the seminar and reflect the experiences and opinions of various attendees. Thus, some of the contents might be incomplete, contradictory, or redundant.

Attendee Mix

The seminar's attendees were 36 internationally renowned researchers as well as high-profile practitioners whose work is closely related to the topic of this seminar. Given that the last years have seen an increased focus on research related to database system reliability, a larger fraction of researchers had significant expertise in this topic as compared to the first edition of the seminar, whose attendees mix had a broader, but less specific expertise. Overall, 25% of the second seminar's attendees joined also the first seminar – note that the first edition was a small seminar with a hybrid format that had only 12 in-person attendees, of which 67% joined the second seminar. A large number of attendees from Europe accepted the seminar invitation, constituting 56% of the attendees. The remaining attendees joined from the US (27%), Asia (14%), and one attendee from South Africa. 36% of the attendees were from industry, which is higher compared to the last seminar, which had only 23% attendees from industry. By increasing their percentage, we aimed to gain more insights into the current practices in the industry. Unfortunately, one of the co-organizers, Alexander Böhm was unable to attend the seminar; we want to thank him for his contributions in organizing it.

Seminar Structure

The seminar lasted for five days and accepted only in-person attendees. We started the first day with a round of introductions, where each attendee could introduce themselves and their interests in a five-minute presentation. The majority of the remaining days were filled with discussions by the individual working groups – we deliberately avoided having presentations on past work to keep the discussion centered on open problems and future directions. While we identified the main topics of interest that helped form the working groups on the evening of the first day, attendees could freely move between these groups and regroup. We reconvened in the larger group once or twice per day, to present and discuss the results of the working groups. The four working groups listed in this report reflect the working groups at the end of the seminar; for example, the first working group on automated testing split into multiple subgroups that discussed various topics during the course of the seminar. In addition to the working groups, we had two tutorials after dinner, as well as an excursion on Wednesday afternoon.

Future Plans

The individual working groups' reports are a concrete outcome of this edition's seminar that future work will build upon. First, working groups indicated their plans to refine their reports as well as follow up on the work to present the results to a broader audience aiming to encourage future work on their topics. Second, we expect individual research groups to take on the challenges identified in the seminar and propose solutions to them. Third, we plan to propose a third edition of the seminar to discuss the progress on tackling the identified problems and expand the scope in terms of reliability. In this seminar, we also want to fill gaps in terms of topics and attendee mix; for example, in this edition of the seminar, we focused our discussions on relational database systems, omitting systems built on other models. As another example, we had no major discussions on the reliability of learned components of database systems.

2 Table of Contents

3 Overview of Talks

3.1 SQLancer Tutorial

Manuel Rigger (National University of Singapore, SG)

This tutorial gave an introduction to SQLancer, which is a popular and widely used tool to automatically test database management systems. The tutorial assumed both a user perspective, as well as a developer one, while also highlighting limitations that could be addressed by future research.

3.2 Informal Proofs of Correctness for Lock-free Algorithms

Russell Sears (Crystal DB – San Francisco, US)

We held a tutorial on informal proofs of correctness for a simple, but powerful class of lock free algorithms. The work has been open sourced, and feedback from the seminar attendees improved the documentation and provided us with references to related academic work on software correctness and testing.

The library is available here:
`https://crates.io/crates/atomic-try-update`

The atomic-try-update library makes it easy to correctly implement your own lock free data structures. In addition to the base primitives, we provide a few example data structures that you can use directly, or that you can use as a base for your own application-specific algorithms.

Typical use cases for atomic_try_update include implementing state machines, building simple resource allocators, initializing systems in a deterministic way using "fake monotonicity", accumulating state in stacks, and using the claim pattern to allow concurrent code to enqueue and then process them sequentially.

Unlike most lock free libraries, we make it easy to compose the above in a way that preserves linearizable semantics. For instance, you implement a lock free state machine that tallies votes as part of a two phase commit protocol, and then combine it with a stack. The resulting code would add information about each response to the stack and then process the result of the tally exactly once without resorting to additional synchronization such as mutexes or carefully ordered writes.

By "linearizable", we mean that any schedule of execution of the algorithms built using atomic_try_update is equivalent to some single-threaded schedule, and that other code running in the system will agree on the order of execution of the requests. This is approximately equivalent to "strict serializability" from the database transaction literature. atomic_try_update provides semantics somewhere between those of a transaction processing system and those of a CPU register. We chose the term linearizable because it is more frequently used when discussing register semantics, and atomic_try_update is generally limited to double word (usually 128-bit) updates.

From a performance perspective, atomic_try_update works best when you can have many independent instances that each have low contention. For instance, using a single atomic_try_update instance to coordinate all reads in a system would likely create a concurrency bottleneck. Having one for each client connection probably would not. This means that you should stick to other, more specialized algorithms for things like top-level event queues and other high-contention singleton data structures in your system.

4 Working groups

4.1 Working Group on Benchmarking

Lawrence Benson (Hasso-Plattner-Institut, Universität Potsdam, DE), Carsten Binnig (TU Darmstadt, DE), Federico Lorenzi (TigerBeetle – Cape Town, ZA), Danica Porobic (Oracle Switzerland – Zürich, CH), Tilmann Rabl (Hasso-Plattner-Institut, Universität Potsdam, DE), Anupam Sanghi (IBM India – Bangalore, IN), Russell Sears (Crystal DB – San Francisco, US), and Pinar Tözün (IT University of Copenhagen, DK)

Motivation

Standardized benchmarks are crucial to ensure a fair comparison across systems. Furthermore, they serve as a testing aid and help find bugs in systems.

In the database community, Transaction Processing and Performance Council (TPC) [43] has been standardizing benchmarks covering application domains from OLTP, OLAP, Big Data, IoT, AI ... Yahoo Cloud Serving Benchmark (YCSB) [16] has been quite popular for transaction processing, key-value stores, and the cloud. The Linked Data Benchmark Council (LDBC) specializes in graph analytics benchmarks [2].

While extremely valuable, these benchmarks all present a static scenario, where the workload is well-defined and known in advance. As a result, data management systems get fine-tuned for a particular benchmark before they are run with that benchmark. Therefore, the presented results often times do not reflect the behavior of these systems in, typically way less predictable and way more dynamic, real-world settings.

Furthermore, it takes time and effort to standardize a benchmark. With the fast-pace that the data-intensive systems and applications evolve today, it is difficult to generate a standardized benchmark to cover a variety of real-world use cases in a timely manner. This often times lead to complaints about the standardized benchmarks not being representative.

To address these issues, we would like to present a complementary approach to the current standardized benchmarking practices by introducing periodic themes and an element of surprise.

Surprise Benchmarking

The principles of the *surprise benchmarking* is similar to the principles of the ACM SIGMOD Programming Contest. More specifically, we would like to establish periodic (monthly, quarterly ...) benchmarking rounds, where a theme for the round will be announced ahead of time such as "OLAP", "key-value store", "UDFs", "resource-constrained environment", etc. There may or may not be a description of the schema, data, a few expected queries, target hardware setup ... During the actual benchmark run, an element of surprise will be added such as ad-hoc queries, new table addition, data distribution changes ...

At each round, we will provide a baseline using a well-known data management system, e.g. Postgres. There will also be a different set of metrics to focus on at each round, in addition to the more traditional throughput and latency metrics, such as energy-efficiency, hardware utilization, monetary cost ...

On the one hand, the surprise benchmarking would be to eliminate the overly-tuned nature of current standardized benchmarking practices. On the other hand, it would help to accumulate a wider variety of standardized benchmarks over the years covering a more diverse set of data-intensive applications and their real-world deployments.

Surprise Categories

We categorize the element of surprise into three based on the intended impact on the system under test. We expect each benchmarking round to have a surprise covering each category. The level of surprises will systematically vary during the benchmark runs; i.e., having several runs with different % of surprise queries. However, how we vary the mix of surprises or whether we mix the surprises across different surprise categories may change round to round.

On-road

This type of surprise represent cases that are supposed to be meaningful for the systems under test. In other words, these are the queries or scenarios the system was designed to cover, and they shouldn't break the system. For example, ad-hoc queries similar to, but not the same, the TPC-H benchmark queries for a system specifically designed for OLAP workloads. Similarly, introducing data or access skew that mimic real-world data distributions as the surprise would fall under this category.

These types of surprises are also the ones that are the most suitable for auto-generation. As part of this work, we plan to develop / utilize the automatic query generation techniques as much as possible.

The goal here is to observe how the system behavior changes compared to fine-tuned standardized benchmarking scenarios.

Wrong-road

In contrast to the previous category, this one represents scenarios that are not the intended use of the system under test. They are a misuse of the system, but the intentions of the user are not harmful. For example, using a SELECT * to get all the data than putting the data through many UDFs instead of doing some data processing, such as filtering data, with SQL; or instead of using JSON support in a system, casting the data to string and performing regex operations on the string would be misaligned uses of a database system.

While not as suitable for automation as the above category, we will still investigate auto-generation of scenarios for these types of surprises.

The goal here is to observe how the system handles the performance impact of such misaligned scenarios and possibly whether it corrects them on its own.

Off-road

The final surprise category represents scenarios that aim at breaking the system by driving it to the edge. For example, introducing a recursive query to bloat the memory resource needs while running that query, introducing extreme data skew, cutting down the hardware resources on the fly by introducing a heavy stream of collocated workloads would fall under this category.

While some automation is still possible with this category, it is the one that requires the most careful crafting, and hence manual effort.

The goal here is to observe the robustness of the systems under extreme scenarios.

Round 0

Before making surprise benchmarking in reality, we devised a plan for an initial test run, *Round 0*. The goal with this round is to establish the methodology and the framework to perform the surprise benchmarking, in addition to evaluate our vision for it. Therefore, Round 0 will be an example run orchestrated by us using the database systems picked by us. We will also be building the preliminary infrastructure for keeping a leaderboard for the benchmark rounds. We plan to share our experiences and findings during Round 0 in the form of a paper, e.g., in DBTest workshop.

To ensure that we focus on developing the bare minimum requirements for the surprise benchmarking methodology and framework, we will keep the benchmark itself as simple as possible. In other words, we would like to start from existing well-established benchmarks. Thus, Round 0 focuses on traditional OLAP workloads and takes TPC-H benchmark as the basis. For the systems under test, we pick Postgres, MySQL, DuckDB, Umbra, and ClickHouse.

4.1.1 Round 0: Surprises

During the real benchmark run, we will complement TPC-H queries with previously unseen queries corresponding to each surprise type described in Section 4.1.

Here are a list of possible surprises that we can create:

Surprises on the Query Level

- Joins on non FK-PK relationships
- Queries with recursion
- Queries with cyclic joins
- Queries with UDFs
- Large SQL strings (multiple MBs)
- Queries that stress memory (intermediates that need to spill)
- Non-equi joins
- ...

Surprises on the Schema Level

- No FK-PK in schema
- Extremely many tables
- Non star/snowflake schemata
- ...

Surprises on the Data Level

- Heavy hitters for joins
- Non-uniform / correlated data
- Size of the data

4.1.2 Round 0: The benchmark procedure

We require databases to support SQL-92 (or similar).

We'll provide a sample data generator and workload 30 days before the run. There will be some sort of leaderboard, and each contestant will be money (cloud credit) limited. Entries will be a deployable setup (either in the cloud or on a dedicated test server) in a format such as Docker or Kubernetes.

We'll include a data loading procedure, with a clean CSV or other file that contains the table data. The databases will have a bounded amount of time to load the data; e.g., set to 10x longer than whatever our reference implementation takes.

Round 1

We plan to run the first round of the real surprise benchmarking around the same time with our paper presentation / publication for the Round 0. The benchmarking run will be open to anyone, but we will specifically reach out to the systems we tested during Round 0 to enter Round 1 themselves.

Long-term plan

We think of starting with quarterly rounds.

We would like to base the rounds on the existing benchmarks as much as possible till the benchmarking rounds become more stable to keep the work around running these benchmarks low.

To check the correctness of the output of the benchmark queries, we will double-check and use the results generated by Postgres. Furthermore, we plan to use Postgres as the baseline during the benchmark rounds, since it is a highly mature and popular database system.

4.1.3 Themes to consider

- OLAP + unseen queries
- OLTP + unseen transactions
- OLAP + add new / many tables
- OLAP + resource-constrained hardware
- OLTP + multitenancy / collocation
- Key-value stores + data skew
- Vector databases + different data characteristics
- Data loading & new table creation
- Cloud (with all of the above)
- Full surprise with mix & match
- Multimedia analysis
- Poorly factored microservices

4.1.4 Metrics to consider

- Throughput
- Latency and latency distribution
- Energy
- Carbon footprint
- Cost (bare-metal vs cloud)
- Resource consumption (memory, disk, CPUs)
- Micro-architectural (IPC, cache misses, etc.)
- Robustness
- Scalability
- Availability

Challenges

- ***Automation*** for both the generation of the surprises and the benchmark runs.
- What would be the ***incentive*** for companies and academics to run and/or enter these benchmark rounds?
- What would be the ***hardware infrastructure*** to support the benchmark runs? Who would provide it? Furthermore, when we would like a surprise hardware element, how much of the hardware infrastructure should be revealed beforehand?
- How do we identify and quantify some of the less traditional ***metrics*** such as robustness, carbon footprint, etc.?

Addressing all these challenges will be crucial for the success of the benchmark. While we aim to tackle these as much as possible with Round 0, some challenges have to be resolved as we have more rounds of benchmarking. For example, we won't be able to automate surprise generation across many workloads and deployment on different types of hardware infrastructures starting from Round 1.

4.2 Working Group on Transactions and Concurrency

Wensheng Dou (Chinese Academy of Sciences – Beijing, CN), Adam Dickinson (Snowflake Computing Inc. – Seattle, US), Burcu Kulahcioglu Ozkan (TU Delft, NL), Umang Mathur (National University of Singapore, SG), Everett Maus (Google – Seattle, US), Stan Rosenberg (Cockroach Labs – New York, US), Gambhir Sankalp (EPFL – Lausanne, CH), Caleb Stanford (University of California – Davis, US), and Cheng Tan (Northeastern University – Boston, US)

This working group focused on the challenges involved in testing and diagnosing transactional database systems, which support complex SQL operations and varying consistency and isolation levels, in a concurrent and distributed setting.

We investigate methods to evaluate the transactional correctness of database systems, i.e., whether the database system satisfies its guarantees under all possible execution scenarios. We excluded problems related to non-concurrent database testing problems (such as ensuring individual or non-concurrent queries, reads, or transactions are correct) and instead focused on problems which are specific to concurrent transaction workloads.

Discussed Clusters

Our discussions identified problems of interest in this space, and found that they can be divided into 3 major clusters:

1. Concurrent test oracles: how can we design test oracles to verify concurrent transactions?
2. Coverage metrics and test generation: How can we identify coverage metrics for testing, in order to generate minimal and sufficient test cases?
3. Bug and failure reproduction: After bugs are found (either by users or in production), how can we effectively reproduce, root-cause, and diagnose them?

In each cluster, we identify the state of the art in practice, problems to be solved that we view as open, related work, and potential solutions.

In addition to the three clusters, there are other dimensions of interest. For the target database system, it can vary from key-value stores (at the simplest level), to relational databases, to supporting client-level transactions. Consistency and isolation levels are another important dimension, ranging from strong sequential consistency to weaker consistency models or only eventual consistency. Problems can also be classified based on the type of bug that is considered: safety bugs such as data races and race conditions; violations of serializability, linearizability, and atomicity; deadlock freedom; liveness violations such as non-termination; and performance bugs. Many of these bugs manifest concretely as either *wrong results* returned to the user, *data corruption* in the database itself, or *crashes* (which is the best case scenario of the three).

Cluster 1 – Concurrent Test Oracle

4.2.1 Motivation

It is notoriously hard to test concurrent problems in databases as they (the bugs/anomalies) rely on various inter-transaction interactions, transaction interleavings, and transactional concurrency guarantees (isolation guarantees promised with the database systems).

Today's approach is to test a database by repeatedly running concurrent workloads, hoping to cover more interleavings, transactional interactions, and database internal schedules. However, given an isolation guarantee (e.g., serializability), the valid states are huge; then, how to examine if a given state is valid is challenging. Note that this is straightforward if a database provides linearizability and outputs a commit timestamp – we re-execute transactions in the order that they committed at, and can check the database state at each point as is done by Spanner at Google. The problem is more complex (from a algorithm complexity point of view) for some weaker isolation levels (e.g., non-strict serializability and snapshot isolation) and for the databases without accurate commit timestamps.

So, we form this problem into the *concurrent test oracle problem*: given some workload of concurrent transactions, how to validate that the responses and the final state of the database is correct with respect to the isolation and concurrency specification.

4.2.2 Batch Formalization

The batch formulation assumes we have executed some set of concurrent workloads, and now want to validate that the outcome was correct.

Given:

Input:

- $t_0 \leftarrow$ start time
- $t_1 \leftarrow$ end time
- {transactions, ...}
- database state at t_0
- concurrency + isolation specification

We assume input state of the database is valid.

The transactions are assumed to contain the sequence of actions they executed (reads, writes, mutations, but also queries and DML statements), the transaction's start and end times (synonymously: arrival / commit time), and the results of reads or queries. (Optionally, we believe it is reasonable to also require the actual values written, if it simplifies the problem.) Note that we require actions to be atomic. This is identical to the standard way of describing concurrency / isolation models for reads / writes, but we also require things such as a query (which may touch multiple tables) read from the same version of the database (and similarly, DML may read/modify/write multiple rows).

For the batch case, we want to be able to validate that the database state at t_1 is valid after the transactions provided have all executed, and that the values read (or queried) by all transactions are correct at each point.

4.2.3 Incremental Formalization

The incremental formalization is aimed at the problem of validating transaction outcomes in an "online" way – as part of an ongoing test, for cases where the batch validation case may not scale (e.g. where hundreds of thousands of transactions might be run over the life of a test, but the number of transactions running concurrently might be quite low).

Given:

Input:

- $t_0 \leftarrow$ start time
- $t_1 \leftarrow$ end time
- transaction to validate starting at t_0 and ending at t_1
- {transactions + metadata, ...} all transactions with overlapping start or end times with the transaction to validate
- database state at t_0
- concurrency + isolation specification

We assume the database state at t_0 is correct. The transactions and metadata are similar to above.

Given this, we want to be able to do one of a few (largely equivalent) things:

- Determine if the actual database state at t_1 is consistent, and that the read/query results from that transaction are valid
- Compute the set of all possible database states at t_1 and the set of all possible read/query results from the transaction

4.2.4 Problem

We define a transaction as a sequence of SQL statements, decomposable into read and write operations. A set of all possible executions of transactions forms a schedule. Thus, an isolation level can be seen as a set of constraints, I, over all possible schedules. We say that a schedule is allowed under an isolation level if it adheres to all constraints in I. E.g., serializability isolation is a set of constraints which ensures that every schedule corresponds to a serial execution of transactions.

There are two mainstream approaches to verifying an isolation level, namely black box and white box oracles. A black box oracle observes only external state changes, same as a database client; i.e., results of individual operations are invisible until a transaction is committed. Conversely, a white box oracle can observe internal states; i.e., results of read/write operations. Each approach is fraught with scalability and expressiveness challenges. The latter arises when transactions are composed of SQL statements. Read and write effects of a SQL statements are in some sense a prerequisite for expressing (transaction) conflicts. Scalability challenge is a function of transaction size (i.e., number of operations per transaction) and transaction throughput (i.e., number of committed transactions per second). Both online an offline approaches exist. An online approach is desirable especially if it yields a mechanism whereby a transaction in violation of the isolation level can be aborted. For practical reasons, e.g., high transaction throughput, offline verification remains a popular approach.

4.2.5 Challenges

There are multiple challenges to address the concurrent test oracle problem in practice.

Challenge 1: efficiency and scalability

Checking efficiency is crucial in practice. Some of the problem variants have been proved to be computational expensive to check; for example, black-box checking serializability and snapshot isolation is known to be NP-Complete. Even for the white-box checking isolation levels that have polynomial checking algorithms, it is unclear if the checking can be efficiently implemented to catch up with the throughput of today's databases.

In practice, a DBMS is expected to meet its consistency/isolation guarantees, and is expected to support high levels of concurrent usage over long periods of time. In practice, this is often checked by long-running tests that aim to provoke a wide array of behaviors over a long (hours to days) length of time.

Challenge 2: beyond reads and writes

The state-of-the-art test oracles almost all work on the transactional key-value model; that is, the available operators are reads and writes to a key once at a time. However, in practice, real-world key-value stores have many advanced operators, including range queries, max, min, and many other aggregation operators. In addition, we barely know how to handle concurrent SQL statements in the concurrent test oracle problem.

Challenge 3: providing evolvability and debuggability

Developer friendliness is also a strong requirement. In particular, if the DBMS itself is extended (e.g. new features are added to the SQL dialect the DBMS supports) then either the underlying solution should not need new work or any extension required should be straightforward for the average developer. For example: If adding support for a new aggregate, such as MIN/MAX, to the test system requires significant work to update an SMT representation, it's unlikely to be a viable solution for industry use.

Similarly, the debuggability of a solution would be important to allow issues found by these systems to be reproduced, understood, and addressed by developers.

4.2.6 Existing approaches

Differential testing

For deterministic (statement-by-statement execution, that is, we can obtain a deterministic execution trace for each statement in the concurrent transactions) transaction test cases, we can compare the transactions' execution results between database systems. However, differential testing faces some issues. First, it cannot compare concurrent transaction executions. Second, different database systems may contain inconsistent transaction semantics, and differential testing can report false positives. Third, different database systems support different isolation levels, which cannot be compared together.

Infer transaction execution result

For some simple and deterministic transaction test cases, we can potentially infer the execution results of each statement in the transactions according to their transaction semantics. This approach can also have some limitations. First, it cannot infer the test oracle for concurrent transactions in which we cannot obtain their statement-by-statement execution. Second, it cannot support some complex SQL queries, e.g., aggregation functions (may not be important).

Key-Value Store Oracles.

This problem has largely been solved for simple key-value stores (i.e., supporting only point reads/writes). However, supporting the more complex semantics that languages like SQL support is an open problem. (Consider that the values read from one table may depend on the rows read from another table within the same query.)

Cluster 2: Coverage Metrics and Test Generation

4.2.7 Current industrial practice

A common way to test the concurrency/transactional correctness of a database is to run a set of concurrent transactions and observe that the database behaves correctly. However, it is non-trivial to determine what cases uncover net-new behavior in the system versus simply validating the same case repeatedly.

The current state of practice in this space tends to simply run a set of tests repeatedly (either for a test budget of a certain amount of compute time or a certain number of test executions) with no clear indication of coverage. This does not provide any confidence to the practitioners about the effectiveness of their testing strategy, leaving the following questions unanswered:

1. Given a particular test case (e.g. a set of concurrent transactions) at what point will running more tests stop providing additional information about the system's behavior?
2. Given two test cases and information about the events that occurred, do they exercise redundant behavior?
3. Running a set of text executions, at what point do additional test cases/executions not provide information about the system? (At what point has this metric been saturated?)
4. At what point have these tests covered enough of the system that we can be confident in the overall concurrent behavior of the system?
5. Is it possible to automatically construct net-new test cases that provide new information about the concurrent behavior of the system?

4.2.8 Problem Statement

To address these questions, we would like to be able to determine:

Defining Coverage How can we define proper coverage metrics that can reflect developers' intuition of test coverage?

Measuring Coverage Given a particular set of test cases, how can we efficiently measure the amount of exercised behaviors of the database systems?

Detecting Saturation Running a set of text executions, at what point do additional test cases not provide information about the system? At what point has the coverage been saturated?

Coverage-Guided Test Generation How can we automatically construct new test cases that provide new information about the concurrent behavior of the database system?

4.2.9 Coverage Metrics

The set of covered behaviors in database systems depends on both (i) test input transactions, which may trigger operations on different replicas and partitions (data-centric behaviors), and (ii) concurrent interactions and partial failures in the database system (concurrency-centric behaviors). The set of transactions running in the test case may trigger different features of the database system exercising different system behaviors (optimization engines, rollbacks, failure recovery, etc.). On the other hand, the same set of transactions can trigger different behaviors of the system, depending on the interleavings of the concurrent interactions in the system (e.g., communication between database replicas, the interleavings of the faults w.r.t. the communication).

Therefore, an ideal coverage metric should be able to capture the coverage in both aspects and also how novel behaviors in one aspect trigger novel behaviors in the other aspect. Even when a given set/sequence of transactions has the potential of exposing subtle behaviors and bugs in the system, not all interleavings may be able to expose them. Likewise, simply exploring the entire space of concurrent interactions between a given set of transactions may not exhaustively exercise features that may otherwise be exposed by different transactions. Indeed, a subtle choreography between transactions and their concurrent interactions is often required to expose bug inducing behaviors.

The existing coverage metrics widely used in practice (such as line or branch code coverage) does not capture the temporal order and the interactions between the transactions. It remains an open research question to define proper coverage metrics to address the problem in the database systems settings running concurrent transactions.

4.2.10 Measuring Coverage

Given the insufficiency of traditional coverage metrics like line and branch coverage, more exhaustive metrics that distinguish temporal behaviors may be more appropriate in our setting. However, the precise granularity and level of abstraction may affect the efficiency of determining the amount of coverage that has been achieved. The efficiency of coverage tracking also crucially affects the runtime of the testing campaign and, thus, the number of behaviors covered within the same testing budget. We think a thorough investigation of coverage metrics that achieve different tradeoffs between abstraction (i.e., does not distinguish between similar inputs/interleavings) and efficiency of tracking coverage may be required to assess their suitability to different settings.

4.2.11 Detecting Saturation

While measuring coverage of a set of tests provides information about the explored set of system behaviors, it does not indicate how much of the possible system behaviors have been explored.

To provide confidence in the testing strategy, the coverage metric should quantify the set of (all) possible system behaviors and how much of those are covered by a set of test executions seen so far. Such a quantification has the potential of providing concrete feedback to developers about how much more computational resources to continue to dedicate and whether or not to terminate the exploration of new behaviors in case the coverage is close to saturation and the marginal utility of continuing is expected to be low.

4.2.12 Coverage guided transaction test generation

The coverage information can be used to guide the generation of new test cases to extend the search to increase coverage. In order to expose concurrency-centric behaviors (in conjunction with other data-centric behaviors), we envision that *scheduling hints* and *hints on partial failures at runtime* can increase the effectiveness of otherwise vanilla transaction-based tests. On the one hand, no control over the scheduling and the injection of partial faults may empirically resemble naive random testing with poor effectiveness. On the other hand, full scheduling hints that can control precise interleavings of processes can be impractical. Further, retrofitting controlled testing into existing large-scale testing frameworks is likely to be not ergonomic and may not be adopted in practice.

The space of granularity of data-centric and scheduling hints should be investigated to build effective and ergonomic guidance methodologies for test generation. The feedback information can then be used in a similar approach to popular feedback-driven fuzzing methods.

Cluster 3: Bug and Failure Reproduction

Suppose that a system has encountered an unexpected bug or crash – either in a complex test or in production. We have the existence of the bug, the logging provided by the system prior to encountering the bug, and any other diagnostic information collected on failure. How can we re-create the sequence of events that led up to this bug?

In our discussions, we identified four major questions that we would like to see more work addressing. This section contains a summary of current industrial practice, a discussion of the four identified questions, and related areas of work.

4.2.13 Current industrial practice

Our industry attendees emphasized that this is a common scenario in practice, and it can be very difficult to reproduce failures. The current state-of-the-art is often simply rerunning a test a large number of times (e.g., 1000x), potentially after augmenting the system with additional logging, a slightly modified configuration, runtime sanitizers, or with changes to the system or test workload to increase the likelihood of perturbing the issue. Due to the lack of support for more targeted reproduction techniques, each test run can require running on 10s to 100s of virtual machines, and the time to run each test takes anywhere from a minimum of 30 minutes to run to a maximum of about 16 hours.

Due to all of these challenges, *known* bugs in production databases can remain undiagnosed for years. For example, one anecdote described a transactional database system with 6 long-standing concurrency-related bugs in production, and reproducing a single one of these bugs in a test environment and subsequently fixing it eventually took 2.5 years.

Whether it is more important to reproduce bugs found during testing vs. bugs found in production, it was answered that both are very useful, but roughly 20x more bugs are found during testing vs. in production. Therefore, new techniques that only work during testing would still be useful, even if the overhead is too high to be deployed in production.

Finally, some industries are leveraging systems for deterministic record-and-replay (see related work below), such as FoundationDB's testing framework [56], Megastore at Google [7] and Hermit at Meta [37]. See also Thomson and Abadi's 2010 paper making the case for determinism [50]. However, there was a feeling that these techniques may not be mature enough to scale to the distributed setting and reproduce transactional database bugs at scale. They require both an upfront design cost and runtime overhead, both of which can be prohibitive, especially when running on multiple machines.

4.2.14 Problems identified

Q1: Reconstructing the system trace

Suppose that some system events and failures are logged, but others are not. Given the set or trace of logged events, how can we reconstruct one possible system execution trace which is consistent with the log?

Note that there may be multiple system execution traces that are consistent with the log; we only need to find one of them in order to demonstrate a bug.

Approaches to the problem can be placed on a spectrum, trading the amount of additional logging required with additional effort in reconstructing the execution:

- On one end, logging every event in the system with a trusted clock (faults, scheduling, message ordering, message deliveries, internal non-determinism) can allow for viable reproduction of the explicit system state, but imposes a large burden on developers to explicitly track any choices that affect state as well as on the logging framework to store and manage the large amount of information.
- On the other end, minimally logging transactions, start/commit times, and the final state can allow the use of an oracle as in Cluster 1 to produce a set of viable executions explaining the behavior. However, the set of executions to explore in this scenario may be too large to reasonably compute, given that logs leading up to a failure may contain thousands to tens of thousands of transactions.

Q2: Amount of logging

The discussion above for Q1 raises the following central question: what is the *minimal* set of information that needs to be logged in order to answer question 1? That is, what is the minimum amount of logging which is sufficient to reconstruct the system state at time of failure, within a given computational means?

Here, it is not necessary that the logged information allows us to reconstruct the execution uniquely; it is sufficient to be able to reconstruct an execution leading up to the final state within a small enough search bound from the input logs. The logged information is then used to filter the space of possible executions.

One caveat that makes this difficult in practice is that sometimes when logging as added, it can make it impossible to reproduce some states (as logging can affect the timing of events).

Q3: Controlling environmental non-determinism

How can we ensure that the environment simulated in a controlled environment matches the production environment? In other words, every behavior that is possible in production must be possible in the simulated environment.

Simulated environments, including e.g. record-and-replay tools as well as fault injection tools, provide a partial solution to the failure-reproduction problem, albeit with significant overhead. The goal of such tools is to provide a highly controlled testing environment, wherein all non-determinism choices are controlled and determinized by the environment. However: 1) the controlled testing environment may not faithfully reproduce the behavior and semantics of the production environment, and 2) the determinization of behavior may preclude discovery of traces that reproduce the failing behavior.

Q4: Retrofitting determinism

How can we add determinism guarantees onto an existing database system that was not built to support deterministic execution?

The challenge here is a little different than deterministic replay; the question is about making the database implementation itself deterministic. For the distributed case, however, we also have to ensure determinism of the distributed aspects through, e.g., simulating the network and injecting controlled node failures.

4.2.15 Related Work

Several lines of related work are relevant to the problems identified above. *Flaky tests* (i.e. tests that may fail nondeterministically), are extensively studied in the software engineering community, (e.g., [35, 8, 21, 31, 42]). Bugs related to concurrency, distribution, consistency, and isolation levels often manifest as flaky tests. Work on *debugging big data applications* is also relevant; the most prominent tools in this space include Inspector Gadget [39] and BigDebug [27]. Finally, *record-and-replay* tools, which include Mozilla RR [38], Hermit [37], and DetTrace [36], provide controlled environments for determinized execution and debugging.

4.3 Working Group on Query Languages and Debugging

Denis Hirn (Universität Tübingen, DE), Moritz Eyssen (Snowflake – Berlin, DE), Tim Fischer (Universität Tübingen, DE), Torsten Grust (Universität Tübingen, DE), Muhammad Ali Gulzar (Virginia Polytechnic Institute – Blacksburg, US), Hannes Mühleisen (CWI – Amsterdam, NL), Thomas Neumann (TU München – Garching, DE), and Mark Raasveldt (DuckDB Labs – Amsterdam, NL)

SQL-level Plan Specification

A SQL query provides a *declarative* specification of a computation over relational tables: it is the database system's task (and freedom!) to explore a space of possible algebraic plans that will implement the query. Query authors need not and (largely) *cannot* influence the system's choice of execution strategy. Declarativity is a core viture of the SQL language –

yet, occasionally, the declarative nature can stand in the way, in particular when precise control over the shape and operators in a physical algebraic plan is desirable, for example in plan-centric database engine testing.

The working group on *Query Languages & Debugging* tackled the challenge of formulating algebraic query plans in a human-readable (and thus: human-writable) fashion. Ideally, this plan format could also serve as a means to exchange physical plans across database engines. In the context of database engine testing, precise control over algebraic plan construction can help to ensure that even the most obscure operator constellations and remote engine code paths are covered – a feature that is hard to realize solely in terms of SQL-level query fuzzing.

The primary guiding principle of the working group's discussion was pragmatics: we settled on (a carefully selected subset of) *SQL itself* as the format in which such algebraic plans are to be written and communicated. Refer to Figure 1 for one sample instance of such a SQL-encoded physical algebraic plan:

- We use a common table expression (CTE) to isolate SQL queries, each of which realize the semantics of a given algebraic operator: for example, query `INDEXSCAN_1` reads columns `a` and `b` from table `r` while evaluating the predicate `a = 42` $\wedge$ `b <= 5`, while query `TABLESCAN_1` scans columns `c` and `d` of table `s`.
- Naming conventions are used to communicate specific algebraic plan operator kinds: `INDEXSCAN_*` advises the engine to employ an *index scan* (here: over a (compound) `a,b` index), while `HASHJOIN_*` represents a *hash join* over given build and probe inputs (see the conventionally named `build` and `probe` row aliases in the query's `FROM` clause).
- CTE-provided query names are then used to assemble tree- or DAG-shaped complex query plan. The algebraic plan encoded by the SQL query in Figure 1 is shown in Figure 2.
- The family of admissable physical plan operators is readily extended: we require a canonical SQL-based formulation of the operators' semantics as well as a recognizable naming convention for the operator itself (as well as its n inputs if the operator is n-ary). We are convinced that a rather small SQL subset will suffice to encode the behavior of even the most exotic algebraic operators.

We envision a query-specific pragma or configuration setting – possibly embedded in a "magic comment" like `--#ENCODED_PLAN` or similar – to switch the database engine into plan-decoding mode. In this mode, the engine detects an encoded physical operator based on (1) its CTE naming convention and (2) a match against the operator's known canonical SQL AST pattern. Table and row aliases are used to wire the physical plan tree.
The rationale of the working group's proposal is as follows:

- Relational already engines come with the required facilities to parse, represent, and serialize SQL queries. No extra plan parser, validator, or pretty-printer needs to be implemented.
- Likewise, query authors and engine developers have already mastered the skill to read and write SQL queries and thus plans. No new syntactic oddities are to be learned and the cognitive overhead is minimized.
- This SQL-based plan format will continue to work even if a database engine only implements a selection of the algebraic operator patterns: CTEs that were not detected as a known algebraic operator can still be wired into the overall plan tree. Their evaluation will yield an intermediate result that can be consumed by the residual plan.
- Indeed, *any database engine* – including those fully unaware of our operator naming and encoding conventions – will still be able to evaluate such a SQL-encoded plan. In a testing environment, the query result may be used as an oracle that provides the expected outcome of the algebraic plan.

```
WITH
INDEXSCAN_1(a,b) AS (
  SELECT a, b FROM r WHERE a = 42 AND b <= 5
),
TABLESCAN_1(c,d) AS (
  SELECT c, d FROM s
),
HASHJOIN_1(a,b,c,d) AS (
  SELECT *
  FROM   INDEXSCAN_1 AS build JOIN TABLESCAN_1 AS probe ON (a = c)
),
RADIXSORT(a,b,c,d) AS (
  SELECT * FROM HASHJOIN_1 ORDER BY a DESC;
)
TABLE SORT;
```

Figure 1 A SQL-encoding of the algebraic plan of Figure 2.

The SQL Acid Test Project

Although SQL is an ISO standard, in practice, systems rarely comply *completely* with the standard. Countless semantic and syntactic differences between systems, often of rather subtle nature, are a sad fact of today's SQL reality. This makes it difficult to write SQL that is portable between systems, and limits the ability to apply SQL skills across database engines – practitioners who are able to write SQL for one database system will face challenges when writing SQL for another database system. Similarly, SQL tooling needs to take these different query language dialects into account, which in practice results in tools being created only for a specific dialect or system, rather than widely applicable tools that can be used for SQL systems in general.

In the world of web browsers, this problem has been encountered before, where browsers (such as Microsoft's Internet Explorer IE6) were found to not be truly standards-compliant. The *Web Standards Compliance Acid Test* [44] was created as an attempt to combat this conundrum. The idea was that users could visit a web page, and if the browser was indeed standards-compliant, it would render a smiley face. This was intended to be an incentive for browser vendors to strive for true standards-compliance. After all, if they were not, their users would see that their browser would not be "smiling at them."

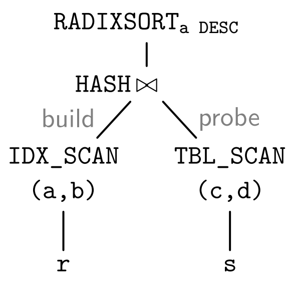

Figure 2 Plan tree of physical operators, derived from the SQL encoding of Figure 1.

```
+-----------------+
|......#####......|
|....##.....##....|
|...#.........#...|
|..#..()...()..#..|
|..#.....o.....#..|
|.#.............#.|
|..#..\...../..#..|
|..#...-----...#..|
|...#.........#...|
|....##.....##....|
|......#####......|
+-----------------+
```

Figure 3 Successfully rendered smiley by a SQL Acid Test compliant database system.

The *SQL Acid Test Project* was launched during this seminar to attempt to create a similar compliance test for SQL database systems. The basic idea revolved around a (large) SQL query that users can execute using the database engine of their choice. If the system is standards-compliant, the query output shows the text-based rendering of a smiley face (see Figure 3). Otherwise, the system will either return an error if parts of the test query are not supported, or parts of the smiley rendering will be replaced with an “`X`” to indicate a deviation from the expected query result.

One problem with the SQL standard is that it is under-specified in many areas, and many behaviors are left up to the implementation. Therefore, while testing for SQL standard compliance definitely is important, it is not sufficient to guarantee that the system will behave properly in all scenarios. For this reason, we have decided to split up the acid tests into two sets:

Compliance tests strictly test if the system complies to the SQL standard.

Convention tests check if the system supports a *sane* SQL dialect as determined by a panel of experts (as of now these have been the members of the working group – a broader community representation clearly is desirable).

test	Db2	Oracle	MSSQL	MySQL	PostgreSQL	Umbra	DuckDB	Snowflake	Spark	GoogleSQL	SQLite3	Presto	Tableau Hyper	MonetDB
01	✓	✓	✓	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	✓	✓	✓
02	ϟ	✓	✗	ϟ	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓
03	✗	ϟ	ϟ	ϟ	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✓
04	ϟ	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
05	ϟ	ϟ	ϟ	ϟ	✓	✓	ϟ	✓	ϟ	ϟ	ϟ	ϟ	✓	ϟ
06	✓	✓	✓	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	✗	✗	✗
07	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	ϟ	✓	ϟ	ϟ	✗	✓	ϟ
08	ϟ	✓	ϟ	✓	✓	✓	✓	ϟ	✓	ϟ	ϟ	✓	✓	✓
09	✗	✓	✗	ϟ	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✗
10	ϟ	✓	✗	ϟ	✓	✓	✗	✓	ϟ	ϟ	✗	✓	✓	✗
11	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	✓	✓	✓	ϟ	✓
12	ϟ	ϟ	✓	ϟ	✓	✓	✓	ϟ	ϟ	ϟ	ϟ	ϟ	✓	✓
13	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	ϟ	ϟ	ϟ	ϟ	ϟ	✓	ϟ
14	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
15	✓	✓	ϟ	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	✓	✓	✓
16	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	✓	ϟ	ϟ	✗	ϟ	✓	ϟ
17	ϟ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
18	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	ϟ	✓	✓	✓	✓
19	✓	✓	ϟ	ϟ	✓	✓	✓	✓	✗	ϟ	ϟ	ϟ	ϟ	ϟ
20	✓	ϟ	✓	ϟ	✓	✓	✓	ϟ	ϟ	ϟ	ϟ	✓	✓	✓
21	✓	ϟ	✓	ϟ	✓	✓	✓	✓	✗	ϟ	ϟ	✓	✓	✓
22	✓	ϟ	ϟ	ϟ	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	ϟ
23	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	ϟ
24	ϟ	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	✓	✓	ϟ
25	✓	✓	ϟ	ϟ	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✓
26	✓	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	ϟ	ϟ	✓	✓	ϟ
27	ϟ	✓	ϟ	ϟ	✓	✓	✓	✓	✓	✗	ϟ	✓	✓	✓
Score	33%	41%	26%	15%	100%	100%	93%	81%	52%	19%	19%	70%	89%	56%

Figure 4 Acid test compatibility matrix as of November 2023. The ✓ symbol indicates that a particular acid test is running successfully on the system. ✗ indicates that the system executes the test, but the result is unexpected. ϟ means that the system was unable to execute the test. The highlighted test cases (▨) are SQL compliance tests.

The *SQL Acid Test* project can be found at `https://github.com/sqlstandardsproject/sqlacidtest`.

4.4 Working Group on Testing "Analytical" Components of Databases

Manuel Rigger (National University of Singapore, SG), Jinsheng Ba (National University of Singapore, SG), Ankush Desai (Amazon – Cupertino, US), Adam Dickinson (Snowflake Computing Inc. – Seattle, US), Wensheng Dou (Chinese Academy of Sciences – Beijing, CN), Stefania Dumbrava (ENSIIE – Paris, FR), Moritz Eyssen (Snowflake – Berlin, DE), Florian Gerlinghoff (MotherDuck – Amsterdam, NL), Hong Hu (Pennsylvania State University – University Park, US), Zu-Ming Jiang (ETH Zürich, CH), Marcel Kost (Salesforce – München, DE), Everett Maus (Google – Seattle, US), Mark Raasveldt (DuckDB Labs – Amsterdam, NL), Andrei Satarin (Google – Mountain View, US), Thodoris Sotiropoulos (ETH Zürich, CH), and Chengyu Zhang (ETH Zürich, CH)

The key problem we are interested in solving is: how can we make sure that a DBMS behaves correctly, without needing to manually enumerate every possible behavior? In particular, *how can we ensure that the behavior of a DBMS satisfies the expected behavior* *in terms of correctness, security, crashes, and performance guarantees*, without requiring a DBMS developer to *expend exponential effort enumerating and testing every behavior and combinations of behaviors*? Although this is a general systems problem, DBMSs are more challenging than many other systems because of their (1) complexity (both of individual features and feature interactions), (2) expected correctness, and (3) persistent state.

Among the seminar attendees, we identified four common themes of interest, which relate to the core challenges of automatic testing:

1. Test case generation, including database and query generation;
2. Test oracles to validate the DBMSs' correctness and other properties;
3. Bug analysis, covering topics such as test-case reduction and deduplication;
4. Transactional testing, which covers aspects of the above challenges, but poses its own special challenges.

Cross-cutting Challenges

We identified various cross-cutting challenges (i.e., challenges that span across two or more of the general challenges that we have identified). We will outline them in this section.

4.4.1 Bug Studies

Various effective automated testing generation approaches have been proposed. However, it is unclear what kind of bugs that they overlook. This question could be studied based on reports in issue trackers or based on customer reports. Specifically, studies could be performed to investigate both characteristics of the features used in the bug-revealing test cases as well as whether existing test oracles could have found them, to identify gaps that could be addressed.

4.4.2 State of the art in the industry

Multiple companies have developed sophisticated testing frameworks. The approaches behind these frameworks are not widely shared, meaning that those unaware of them might reinvent the wheel or implement suboptimal approaches. It would be ideal if companies could share some of their tools or insights; one way forward could also be for academia to conduct interview studies with practitioners.

4.4.3 Interfaces between Existing Tools

Current testing tools typically consist of a database generator, query/workload generator, test oracle, and potentially a component to reduce and deduplicate test cases. These components are generally tightly coupled and it is currently not possible to easily integrate them. For example, it would be difficult to combine the query generator of one tool with a test oracle from another tool. Interfaces could be defined that would allow combining them, similar to existing work on formal verification tools [10].

4.4.4 Query Dialects

In the relational setting, despite the existence of an SQL standard since 1986, a known problem is that many very different variations of it have been implemented in commercial systems [30, 4]. The problem also exists in the non-relational setting. Indeed, for graph databases, the current commercial eco-system is fragmented across dozens of vendors that each implement their own graph query language, as recently surveyed in [9], spurring interest in emerging standards, such as SQL/PGQ [18], which extends SQL with graph queries, and the GQL native graph query language.

Differences between query dialects are a challenge for test-case generation, the test oracle problem, and test-case reduction techniques. For test-case generation, it is difficult to design general generators that both produce valid databases and queries for various dialects, as well as dialect-specific features. For the test oracle, it is difficult to apply differential testing or implement reference engines due to differences between dialects. For test-case reduction, most current reducers are dialect-specific or rely on general text-based reductions.

Test Case Generation

Problem statement. In the context of DBMS testing, a test case consists of an initial database state (schema and data), some amount of workload (queries, transactions, schema changes), and the final expected state (expected results of queries, final database state, etc.). Traditional software development processes involve the creation of manually written test cases to ensure correctness. However, this approach leads to a fixed test suite that, even though it might be over a large area of the input space, still leaves many states unexplored. Therefore, an alternative approach is to generate test inputs automatically and randomly. An important challenge associated with randomized test input generation is: *how can we automatically generate diverse test cases that exercise interesting behaviors in the database engine under test, and thus uncover bugs?*

4.4.5 Existing Techniques

We identify and review the current state-of-the-art testing techniques as follows.

Manually written test cases. These sorts of tests are common across all (industry?) databases and consist of manually curated sets of initial states, workloads, and expected states. The problem with solely relying on manual testing is that:

- The space of feature interactions in a database grows exponentially as new features are added
- It is generally infeasible to enumerate all of the states that all users of DBMS will provoke

User query analysis. Another source of test cases and validation is using actual user workloads – provided by a DBMS consumer, observed in production (for databases that are also hosted by the database developers), or similar. These can be either run in the same environment (for hosted databases) or added to test suites maintained by the database developers. The primary limitations with user workloads are:

- New database features will not be exercised until they are available to the DBMS's users
- Most user's schema, data, and queries are often nonpublic and may not be available to a DBMS developer (for numerous reasons, including legal restrictions on sharing certain types of data)

Randomized testing. Randomized testing is a technique through which test data is randomly generated based on given adequacy criteria/requirements/specifications. The system under test (SUT) is then executed against the test data and the corresponding outputs are evaluated to assess whether they conform to the expected results. The phases of random testing comprise data generation, execution, and evaluation. There are two main approaches to constructing test cases, each with its strengths, limitations and challenges: (1) random test case generation and (2) mutation-based generation.

Random test case generation. Using this approach, test cases (queries) are constructed completely from scratch. A key challenge here is the creation of syntactically- and semantically-valid queries to help uncover deep DBMS bugs. This is typically achieved by implementing a query generator that consults the grammar and the semantics of the DBMS under test. A fundamental limitation is that such query generators are often tailored to specific SQL dialects and DBMSs.

Mutation-based generation. This approach synthesizes queries by modifying existing ones, known as seeds. These modifications include replacements, additions, deletions of tokens, expressions, or complete statements, Mutation-based generation results in grammatically-valid queries by making small changes to existing, valid seeds. This helps exercise different behaviors in the database, while preserving much of the structure and characteristics of the given seed queries. A primary limitation of mutation-based generation is that its effectiveness relies on quality of the available seed programs.

Black-box vs. grey-box query generation. Test case generation techniques can be also grouped into two different categories based on whether the generation process leverages feedback from SUT. *Grey-box test generation* allows guiding the creation of new test inputs by considering feedback from previous executions. This feedback can include code coverage [55], query plans, newly-explored paths and code regions, or the state of SUT. *Black-box test case generation* treats SUT as a "black box" that simply takes an input and produces an output, without knowing anything about its internals.

Existing tooling. Popular query generators for relational databases include the following. SQLsmith [48] is a fuzzer for SQL that can target PostgreSQL by generating random queries from scratch. It however sacrifices complexity for validity, as it only generates one statement in each query, without analysing the dependencies between these statements. Conversely, SQUIRREL [55] can generate queries that contain multiple statements and infer dependencies between them but can produce invalid ones. An emerging challenge is accounting for the trade-off between the complexity and the validity of generated queries. To address this, recent approaches, such as DynSQL [28], combine query generation and execution and use the database state to inform query generation. Similarly, the GoogleSQL Random Query Generator used in testing at Google by Spanner, BigQuery, etc.) uses the latest database state and information about the current contents of the database tables to inform the generation of queries that are more likely to be successful.

As subsequently detailed, limitations of existing tooling include:

- The generated workload may not reflect how real users use the database
- It is challenging to ensure the workload generation tests the DBMS features it is expected to
- Avoiding generating spurious or less interesting test cases requires increasing the sophistication of the generator (e.g. queries that do not parse, type-check, are not valid for the database schema, or will not have any results given the contents of the database)

4.4.6 Open problems/challenges

4.4.6.1 Fuzzing Challenges

To get better coverage of possible inputs to a DBMS and potentially detect more bugs, randomized generation of test workload (for example, a schema, data to populate the schema, and a sequence of SQL queries or database transactions) is a common pattern that finds a large number of bugs.

- Test Case Quality: Given an infinite input space, fuzzers should ideally generate "interesting" queries or workloads, but it is not immediately clear what "interesting" entails. The reason for this is that it is not known beforehand where the bugs hide, hence it is important to test a large share of possible inputs.
- Test Case Complexity: Fuzzers like SQLancer were able to detect many bugs in various DBMSs by constructing rather uncomplicated data and queries. To investigate the potential of finding bugs with more complex queries, e.g., involving a set of complicated join operations, insights into actual bugs that were found by customers would be helpful.
- Feature Interaction & Feature Targetting: A common source of errors is cross-feature interactions. Features in this sense are developer-defined and range from different physical operators to interactions between subsystems to specific code paths taken. It is desirable that a test suite validates interactions between combinations of those features. Most DBMS fuzzing tools do not provide a way to steer generation in a direction that targets a specific set of features. When that capability exists, it is generally not a guaranteed behavior, but instead involves careful tuning of probabilities. Generating queries for a particular feature would especially be helpful in testing new features, which tend to be less well-tested and less mature, and thus a potential source of bugs. Furthermore, by constructing queries that target specific features, such fuzzers could give confidence that relevant parts of the DBMS are tested.

4.4.6.2 Customer-Representativeness of Test Cases

In addition, the goal of testing can influence the desired properties of the generated test workload. For regression tests, a desirable property could be to resemble the workloads of typical customers. Generating these can be challenging since actual customer workloads are often not accessible directly, and accurately modeling them in synthetic workloads is difficult. On the other hand, for teams whose goal is to find as many bugs as possible (even if a customer might never encounter them), a focus on unusual or unexpected inputs, or particularly complex workloads, might be preferable to cover possible edge cases. In practice, there are no tools that provide both, and no tools provide a smooth way to transition between "closer to customer" and "closer to edge cases").

4.4.6.3 Measuring Coverage

As with every testing method, fuzzers need to strike a balance between the test coverage and the cost of testing (time to create and maintain tests and test infrastructure, test execution time, and compute). We want to point out that test coverage here should be understood in a broader sense than simple code coverage. Determining a point of saturation is an open problem – at what point have we run enough tests?

4.4.6.4 Portability

Because different DBMSs come with different SQL dialects and hence differences in syntax and semantics, randomized query generators must often be adapted to be able to generate valid queries that cover all potential features of the system. This involves more work in the development and maintenance of the query generator. An open problem is the development of a portable query generator that can be applied to a variety of DBMSs while keeping the ability to generate complex queries.

An adjacent problem is the lack of compatibility/interoperability between existing query generators (and other parts of a complete test system, as discussed in cross-cutting problems). Currently, ideas from one research group have to be reimplemented by other research groups in order to integrate them into their own solution, and in the industry when these techniques are used, they usually are implemented per-database or per-SQL dialect.

4.4.7 Future Directions

Study of database differences. Another important future direction is to investigate the differences between database engines. This study should answer (1) to the which extent mainstream database engines differ, (2) what are the main sources of their differences, and (3) how do these differences affect the effectiveness of bug-finding tools?

Portable/Extensible query generation. Database engines exhibit distinct differences in their syntax and semantics. To address this challenge, there are numerous proposed approaches:

- *General query generator*: There is some evidence from the industrial partners that highlight that most of the database bugs are found via simple test cases and features, e.g., test cases that only contain `boolean` or *integer* data types. Therefore, it might be sufficient to come up with a general query generator that only covers the core part of SQL, which is commonly supported by databases. This general query generator should be extensible and support configuration options to (1) enable or disable database-specific features, or (2) combine simple test cases into more complex ones [28].
- *Learning-based approach*: A learning-based approach does not guarantee that the synthesized queries are valid. When a semantic or a syntax error is encountered, the approach examines the error message raised by the database engine, and adapts the generation process accordingly. For example, such a learning-based approach stops producing an erroneous combination of features.
- *Encoding database semantics (or part of it) into a specification:* It could be possible to encode part of database semantics through a specification. The benefit is that the constraints and the semantics of each database is described declaratively, rather than being hardcoded into the underlying generation process. Therefore, the input of a future and general query generator could be a specification that describes both the grammar and the semantics of a database under test. In this way, the generator produces queries that are both grammatically and semantically valid with regard to the given specification. For example, in a similar manner to the work of Dewey et al., [19] that relies on *constraint logic programming (CLP)*, one could encode the type system of each database engine into Datalog relations [12]. Based on these relations, we could then produce valid queries that adhere to the typing rules of each database. Notably, Datalog has been the foundational basis of many existing (relational) and emerging (native graph) query languages. As such, another research direction would consist of using it as a system-agnostic basis for specifying the semantics of classes of query languages, ranging from relational (recursive) queries to navigational ones for querying semi-structured data [45, 3] that rely on Regular

Path Queries (RPQs) and extensions (CRPQ, ECRPQ, DRPQ, GXPath, etc). Moreover, due to its generality, Datalog can also encode both syntactic (schema) constraints, as well as semantic ones, i.e., key constraints and functional dependencies. Hence, starting from the specification of a Datalog-based engine, one could generate semantically-aware test cases and leverage these for automated testing. Such test cases could be obtained from the Datalog specification, using property-based testing (PBT) techniques. Moreover, such specifications can also be formally verified, together with the correctness of the generators themselves, for example by leveraging existing plug-ins for automated theorem provers, such as ACL2 [14], and proof assistants, such as Isabelle [11], Agda [20], and, more recently, Coq [40]. Indeed, recent work has shown the promise of using PBT tools, such as the QuickChick plugin for Coq, to *automatically generate reliable generators* for programming language specifications, with high correctness guarantees. The challenges for adopting such methods for database testing are three-fold. The first issue concerns their *usability*, as identifying suitable properties to test, encoding theses, and integrating PBT into real-world database management system architectures is non-trivial. Second, their *extensibility* is also a concern, as such generators would need to be adapted to account for the idiosyncrasies (both syntactic and semantics) of each SUT. Third, there is an *expressiveness vs. performance trade-off* regarding the properties that such formal PBT tools support. An interesting future direction could be to combine such methods with other testing techniques, e.g., coverage-guided fuzzying [32] and combinatorial testing [23]. In addition to Datalog, another recent example is ISLa [49]. ISLa is a declarative specification language that extends context-free grammars and grammar-based fuzzers by constraining the generated inputs (e.g., a variable has to be defined before it is used).

Techniques/Metrics for guiding and evaluating the query generation process. Traditional code coverage metrics, such as line coverage, branch coverage, have been insufficient in effectively evaluating modern testing approaches [34, 13]. There is a need to devise new ways to guide and evaluate query generators. For example, we could use a property vector that indicates whether a certain query is interesting and expressive. Such a property vector could include both static (black-box) features, (e.g., the type and the order of SQL operators), and dynamic (grey-box) features (e.g., configuration parameters, data read/written). It is worth noting that this approach enables feature-aware test case generation. This means that by limiting or expanding the domain of property vectors, we can generate test cases that exhibit specific features, e.g., feature A and B, or feature A and not feature B. This could be useful for swarm testing [26].

Another example of an approach that goes beyond traditional code coverage metrics is inspired by the work of Park et al. [41] on conformance testing of JavaScript. We should investigate whether their feature-sensitive approach is applicable to the database world.

Running queries in different database states. In contrast to compiler testing, where the primary focus is on generating interesting inputs (programs), in database testing, we need to consider both test case generation *and* database state generation. This is because, certain database bugs might be triggered only when the database is in a certain state. Therefore, a future direction should run the generated queries in various database states. A database state includes database schema, logical and physical data, or database configuration.

Test Oracles

One of the core issues with automatically running test cases and automatically finding bugs is checking whether or not the result of a query is correct or expected. There are different approaches with validating that the result is correct.

4.4.8 Existing Techniques

4.4.8.1 Test Suites

Test suites manually specify the test oracle as assertions for a specific test input or scenario [24]. Virtually every DBMS has manually written test suites as part of their testing efforts. They are certainly beneficial to provide fast and reliable feedback to developers on the quality of critical features.

Historically these test suites are hard to reuse across engines:

- they are strongly tied to particular DBMS SQL syntax and semantics;
- they use custom test runners incompatible with other DBMSs or tech stacks;
- alternative test runners are hard to build because test case representation is not specified

4.4.8.2 Differential Testing

Differential testing compares the results of multiple systems and checks them for discrepancies. These systems could be:

- Actual different DBMSs supporting different SQL dialects, which makes their comparison difficult (see the RAGS system by Microsoft [47]);
- A reference engine implemented for that purpose, which can serve as an effective test oracle, but requires significant implementation effort (see more details below);
- the same DBMS, but with different configuration/optimization options.
- Customer query replay [53, 22, 52], where historical customer queries are replayed with a new release or feature enabled. Query results, performance, and cost can then be compared. Queries could be sampled for replay by a combination of random sampling, analysis of feature applicability or plan changes, or cost.
- a previous version of the same DBMS, to prevent regressing in behaviour. Note, that new features can't be tested with such an approach and you carry existing behavior into the future even if it's incorrect. Another variant of this approach is the use of expected results stored with tests [24] or in a database.

Comparing results between two systems is not free of challenges itself. Even very close systems (e.g. previous and current versions) might yield different results due to:

- inherit non-determinism or under specification in the SQL language;
- non-determinism in the query;
- floating point rounding errors due to different accumulation order;
- difference in corner case semantics (e.g. handling of NULL, rounding).

4.4.8.3 Reference Engine

A reference engine is a general-purpose oracle for a database that re-implements the database's semantics in a simpler way, greatly simplifying correct implementation.

Advantages.
- Checks result correctness, rather than just result differences
- Decouples test case generation from validation
- Can validate any supported workload
- Can handle non-determinism and imprecision
- Developers don't need to learn new skills to extend the engine
- Feature support added incrementally during feature development
- Have been extended to strictly serializable workload validation [17]
- Customers could develop and test against the reference
- Has been used for cross-database semantic compliance comparison [6]
- Workload / reference partitioning can improve scale [17]

Disadvantages.
- Large investment to build for existing engine
- Different DBMSs require different reference engines for their dialects
- Requires continuous maintenance to keep up with changes
- Limited scalability in terms of data size, intermediate result size, and query result size
- Limited published research and examples

The reference engine implements either all or a subset of the database API and semantics. For example, some reference engines only support query execution, and some support DDL, DML, indexes, queues, full-text search, etc. It is also possible to reuse some components that will not be verified by the reference, for example, the parser/resolver or scalar functions.

The reference engine should also encode non-determinism, ordering, and imprecision in its values and tuples. This enables the reference to logically return a set of possible correct results and test that the real engine result is in that set. The sophistication can vary from just a special Imprecise value to values supporting epsilons, complex partial ordering, or even multiple logical intermediate result sets.

Open Questions.
- Can we create a general or shared reference? Possibly define an algebra with configurable semantic options and each database family implements a parser and resolver to the shared algebra. Then operators, expressions, etc can be shared when possible.
- Can a reference engine be used to validate weaker transactional semantics than strict serializability?

Examples. Google Spanner has been system tested [17] using a reference engine [25]. ZetaSQL [25] engine's language compliance was also verified using a reference engine [6] (see 6. COMMON SQL DIALECT).

4.4.8.4 Metamorphic Testing

Metamorphic testing generates two related test inputs and checks a relation between their output. For example, many test oracles check that equivalent queries compute the same result.

Advantages include.
- Applicable to DBMSs using different dialects
- Multiple metamorphic relations can be proposed to test different SQL features.

Disadvantages include.

- Missing bugs when both two queries produce the same incorrect results
- Not always possible to cover all features

Logically Equivalent Queries. Generating logically equivalent queries that should return the same result but run a different code path is a way of finding bugs without needing to involve another engine. Examples of this include the work in SQLancer, where queries can be split up into three different queries that are combined through UNION ALL. Similarly, GROUP BY queries can be partitioned along the group, and certain aggregates can be split up (e.g. SUM(x) can have its child tables partitioned and then unified).

One issue with generating logically equivalent queries is that the same code path might be triggered within the same engine so that the logically equivalent queries both suffer from the same bug and produce the wrong result. A potential way of discovering this is to look at e.g. code coverage and verify that the queries actually take different code paths.

Physically Equivalent Queries. Queries that are not logically equivalent might still be equivalent given a specific data set. For example, a query that returns 50 rows, will not return a different data set if a LIMIT 100 is added to the end. Similarly, we can add a filter that has no effect, a filter that filters out rows that would have been filtered out in a later step, or a LEFT JOIN that has no matches on the right side.

Rerunning the Same Query. When running a query on a given schema, the same result should always be produced (except for issues of determinism as discussed later). It can be useful to run the same queries on a database while changing other variables.

Equivalent Schema. We can execute the same query on the same schema, but where the data sources are stored in a different manner. For example, the table can be stored in a different data format (e.g. a Parquet file vs the databases' native format, splitting data into multiple files, etc). We can also run the query over a *view* versus the result of that view stored in a table.

Modifying the Environment. Another variable that can be changed is the environment. For example, we can run the same query on different operating systems or different processors and the same result should be returned.

Toggling Feature Flags. A query can be run with specific features enabled or disabled and the query should still produce the same result in both scenarios. For example, we can enable/disable the optimizer, enable/disable certain types of operators (e.g. hash joins), enable/disable parallelism, enable/disable spilling data to disk, etc.

Nop Rows. We can insert rows into the base table that should be filtered out by the query and not be part of the end result. For example, we can insert rows that should be filtered out by a WHERE clause or by a join condition, or that input a NULL into an aggregate or 0 into a SUM. These rows should not affect the query – allowing us to re-run the query both with and without the nop rows.

Future Problems

4.4.9 Study what the missed bugs are

Various test oracles have been proposed that were effective in finding bugs. However, it has not been systematically investigated what classes of bugs existing test oracles overlook. We believe that systematic empirical studies based on issue trackers and postmortems could shed some light on bugs existing test oracles fail to find. This would be the first step to identify gaps for new test oracles.

4.4.10 Compose or generate test oracles

Existing approaches have proposed many manually-crafted metamorphic relations of SQL (e.g., Ternary Logic Partitioning) as test oracles to detect different kinds of logic bugs. However, there should be many other metamorphic relations in standard SQL and DBMS-specific features, which we have not explored. This raises an interesting research question: How to automatically mine possible metamorphic relations from SQL specification, execution traces of queries, etc.?

From another perspective, a query should return the same result on a logically equivalent data (e.g., the same database with different configuration, and the same view with different physical schemas). This raises another interesting research question: Given a query *q* and database *db* that the query operates on, how to generate interesting equivalent databases on which *q* can still return the same result?

An example might include generating a query which is semantically equivalent to SELECT * FROM T1. Then that generated query could be used to create a view or materialized view V1 which is equivalent to the base table. Further objects could then be generated based on both T1 and V1 to create even more complex test objects which are equivalent to T1.

Assume that we have a couple of equivalent patterns for specific SQL features, is it necessary to compose individual patterns in complex ones? In ideal cases, complex test oracles can quickly trigger more bugs in a single testing process, instead of trying individual patterns one by one.

4.4.11 Coverage with respect to oracles

With many generated queries and many test oracles per query, execution time for the test grows dramatically. It is essential to understand that a particular test oracle provides more (useful) coverage for a given query. It is also important to evaluate if sufficient validated coverage has been achieved by all oracles prior to a release. This could be seen as a prioritization problem. We would ideally want to run all queries with all possible oracles, but this could be prohibitively expensive. Picking first test oracles for query which will yield "better" coverage is crucial.

We also want to evaluate if the oracles we choose can validate all aspects of the database and identify validation gaps that need to be addressed.

Prioritization might depend on

- prior knowledge about coverage, e.g. we want to verify certain oracles (invariants) for majority of the test queries;
- or the query itself, e.g. we expect certain query shapes provide more coverage with certain oracles.

Another approach is oracle-aware test generation. How can we generate queries for a given test oracle to quickly yield "better" coverage?

One potential method to generate oracle coverage for differential testing-based approaches would be to track coverage on each side and identify the different features or code paths covered. Coverage in a reference engine might be achieved similarly, as the difference would exclude any shared code, for example, a shared parser or scalar function implementations.

4.4.12 Designing for Testability

Many current testing tools and their test oracles have been designed independently from the DBMS under test. Certain additional features in DBMS could be useful in testing only and relatively cheap to implement or expose:

- SQL parser as a separate component,
- SQL analyzer as a separate component,
- ability to get metadata about query from the engine, e.g. cost estimate, determinism property, etc. This might include things that can be determined statically or dynamically during execution.
- clearly distinguish between different error types. E.g. syntax, semantics (constraint violation) or internal.
- provide more details for internal errors for tests only. Intention is to help with debugging of failed tests.
- allow to extract variable information from error messages programmatically, instead of using RegEx for that.
- Enforce alternative query plans (all plans vs. choosing other plausible plans dependent on the data distribution; e.g., select top N plans)
- ability to enabled / disable optimization flags. This provides test oracles and aids debugging.
- Self-check options (e.g., `https://www.sqlite.org/pragma.html#pragma_integrity_check`) to expose potential internal errors masked or invariants only enabled in self-check mode (e.g. debug asserts).
- Deterministic execution option
- Deterministic execution/simulation testing
- allow to change table statistics (e.g. histograms) not used for correctness (e.g., often required for MIN, MAX, is NULL). This will help to steer queries into different plans without changing data itself, which might be costly.
- white-box testing of the optimizer (e.g., plan stability). Exposing optimizer API for testing will help with tests which target query optimization tests directly without paying execution costs. Query optimizer is large and important enough component to merit implementation and maintenance costs for this test-only API.

4.4.13 Non-deterministic queries

Queries can be ambiguous and yield different results. Common examples include insufficient ordering with limit/offset, floating point imprecision in aggregation, non-deterministic functions, and runtime errors which are avoided due to short-circuting during execution. This is a problem when test oracles rely on a specific result or compare the results of SQL semantically equivalent operations without modeling this non-determinism. Test oracles would ideally be free of false positives, which is why such ambiguous queries should be identified or avoided. (Or the test oracle should model these sorts of non-determinism, which adds additional complexity to a test oracle.)

4.4.14 Hitting duplicate bugs

Testing tools might frequently hit the same underlying bug. To continue exploring the state space and find additional issues it is valuable to build tools to identify and ignore duplicate issues. This problem is inherently difficult, as deciding whether two bug-inducing test cases are duplicates or not often requires understanding the bug at a source code level. In practice, matchers are used that classify a bug-inducing test case based on an error code, configuration, SQL text, or query plan. Often, constructing these matchers can be time consuming and manual. Some databases have implemented failure signature or feature extraction with automated clustering, which is integrated with bug management to speed up bug detection, provide prioritization data, and accumulate test cases for each issue.

Another interesting direction is to prevent testing tools from repeatedly triggering the same bugs. For example, we could try to guide the query generator not to generate test cases similar to the cases that have already triggered bugs (code-coverage guiding and query-plan guiding can do it to some extent?). It seems like a cross-cutting issue in test-case generation, test oracle, and test-case reduction.

4.4.15 Dealing with Semantic and Runtime Errors

It can often be difficult to avoid statements that result in semantic errors. For example, avoiding an `INSERT` statement to a table with `UNIQUE` constraints might require encoding constraints, which can be complicated (e.g., collation handling with strings). Another common example is generated expressions evaluating to zero, resulting in division by zero errors. Current testing tools typically tackle this in a simpler way, by generating potentially semantically-invalid statement, and annotating the statements using a list of so-called *expected errors*. For example, an `INSERT` statement might be annotated with an expected error "UNIQUE constraint failed". More systematic ways could be explored; for example, the DBMSs could expose different list of expected errors (e.g., syntax errors, run-time errors, or internal errors). For example, this is done by the ZetaSQL engine in code here: `https://github.com/google/zetasql/blob/master/zetasql/compliance/runtime_expected_errors.cc` – a common model for enumerating these sorts of errors and categorizing them would help in creating any general testing solution. Systematically exploring unexpected errors may be another solution for this challenge.

4.4.16 Monolithic/Coupled Testing Tools

Existing testing tools couple their test generation and test oracles, which limits them from finding more bugs triggered by uncovered test case patterns. For example, query partitioning requires that the test case contain predicates to be partitioned. If the test case does not contain predicates (*e.g.*, `INSERT` statements), query partitioning does not work anymore. Decoupling test-case generation and test oracles is not easy, as the test oracles usually require designated test cases to be generated. Using reference engines helps decouple test-case generation and test oracles if the reference engines can process general queries. However, implementing reference engines consumes lots of effort. Another interesting direction is to make testing tools provide interfaces for different test-case generation and different test oracles. The test oracles can choose their suitable test-case generation. It helps the decoupling, but the question is how to design interfaces that are extensible for both test-case generation and test oracles. We may develop a domain specification language that specifies what features are supported by a test oracle.

4.4.17 Non-functional issues

Most existing test oracles focus on logic bugs, which cause the DBMS to compute an incorrect result. However, DBMSs can also be affected by other kinds of issues such as performance issues, issues in cost models, memory consumption, or metastable failures. In addition, for distributed systems, errors could be masked by retries/replication and other techniques, meaning that they do not surface as logic bugs. New test oracles are needed to tackle such issues, while preventing test oracles from being coupled too tightly with the DBMS under test.

Additionally to different non-functional issues, DBMS needs to be further developed and operated. This brings its own set of non-functional challenges. The next step in the development life cycle is deployment. Deployment requires choosing the deployment model, e.g. blue-green deployment, rolling update, full stop and restart, etc. Any chosen deployment approach should be covered by testing to make sure deployment does not introduce issues in production, most notably different versions of a DBMS talking to each other should work correctly.

DBMSs usually have additional features to support operations. These might include backup, restore, various methods of introspection, integration with cluster management for distributed systems and alike. The full list of these features is highly specific to a particular DBMS and environment it targets to run in.

Backward / forward compatibility is another non-functional requirement. Which might extend to file formats, configuration formats, different client versions.

Non-functional requirements are often orthogonal to correctness requirements and test oracles can be reused.

Bug Analysis

4.4.18 Problem Statement

The previous chapters talks about how to generate a large volume of test cases to hit as many bugs as possible. But, because debugging is still a manual task that is done by developers and cannot be automated, dealing with these failing test cases has two major problems.

1. Many different test cases will fail due to the same underlying root cause, making the number of failing tests in orders of magnitudes higher than the number of bugs.
2. The failing test cases are likely to contain very complex queries (and a lot of data) which makes it hard for developers to find the issue.

Ideally the tester only reports a single minimal test case per bug.

4.4.19 Existing Techniques

4.4.19.1 Test Case Reduction Techniques and Tools

We have witnessed various techniques proposed to reduce the complexity of bug-triggering queries and databases, summarized as follows.

1. **String-level manipulation:** This kind of technique involves simple string operations without understanding the SQL grammar. String-level manipulation techniques can be general and relatively easy to implement. The general test case reduction methods like delta debugging [54] can be used for performing the reduction for SQL queries. This kind of technique is simple and already pretty effective in practice [5] but it might cause many invalid queries during the reduction.
2. **Syntax-level manipulation**: This kind of technique tokenizes the SQL queries or parses the SQL queries and operates on the Abstract Syntax Trees (AST). It is a more fine-grain technique than string-level manipulation, which can always generate syntax-valid queries. There are several problems with this kind of technique. First, it is challenging to be

portable to the SQL dialects. Engineering efforts will be required to implement the corresponding parsers and tokenizers. Second, it is still possible to mess up the semantics of the queries, but it often works fine when just removing the leaf nodes or sub-trees on the AST. Third, it may not be able to get to the global minimum by

3. **Semantic-level manipulation**: This kind of technique parses and semantically analyzes the queries to reduce the test cases. It can always create semantically valid queries. The high-level idea is to transform the query into a simplified version while handling the schema and the data dependency. Such transformations are not necessarily to be semantic preserving but need to generate executable queries. It requires extensive efforts to analyze the queries, which may reuse some components of the database management systems, especially the components related to the query optimizations.

The deeper the understanding of the query is (string-level, syntax-level, semantic-level), the easier it is to generate a valid query (syntactically and semantically valid) and to find a global instead of a local minimum. However, at one point the reducer almost reached the complexity of half a DBMS and also is not portable anymore. However, such a reducer can reuse parts of the actual DBMS (if they are generic enough).

Following these techniques, practitioners in industry and academia have implemented diverse tools for various purposes. We collect a set of representative tools as follows.

- **SQLreduce**:[1] This reducer tries to remove tokens by parsing SQL statements into Abstract Syntax Trees (ASTs) and is specific to PostgreSQL. It belongs to the second method *Syntax-level manipulation.*
- **C-Reduce**:[2] This reducer is one of the most commonly used reducers for the C/C++ languages and also works well for SQL statements. It deploys various heuristics to conduct a *String-level manipulation* by gradually removing strings without understanding the semantics of test cases.
- **SQLright Minimizer**:[3] This reducer minimizes test cases by reducing the tokens identified in a customized Intermediate Representation (IR). This reducer was implemented as a component for the database testing work *SQLRight* [33]. It belongs to the method *Syntax-level manipulation.*
- **SQLMin**:[4] Similar to *SQLreduce*, this reducer also minimizes test cases by removing the tokens based on ASTs. This tool was implemented as a component for a database testing work *APOLLO* [29] for finding regression performance bugs. This technique belongs to the method *Syntax-level manipulation.*
- **SQLancer Reducer**:[5] Similar to *SQLreduce* and *SQLMin*, this reducer also parses SQL statements into ASTs and reduce them by dropping the tokens identified on ASTs. This method was implemented as a debugging tool for the database testing framework *SQLancer.* This technique belongs to the method *Syntax-level manipulation.*
- **Percona Reducer**:[6] This reducer adopts several predefined heuristic rules to identify the tokens from SQL statements without understanding SQL semantics. It is included in the toolbox Percona-QA for database quality assurance, and belongs to the method *Syntax-level manipulation.*

[1] `https://github.com/credativ/sqlreduce`
[2] `https://github.com/csmith-project/creduce`
[3] `https://github.com/PSU-Security-Universe/sqlright/blob/main/SQLite/scripts/minimize.py`
[4] `https://github.com/sslab-gatech/apollo/blob/master/src/sqlfuzz/sql_minimizer.py`
[5] `https://github.com/sqlancer/sqlancer/blob/main/docs/testCaseReduction.md`
[6] `https://github.com/Percona-QA/percona-qa/blob/master/reducer.sh`

4.4.19.2 Test Case Deduplication Techniques and Tools

Depending on the failure type of the failing test case, different deduplication techniques can be used. While some techniques are completely unaware of both the test case (query and dataset) and the failure, others use the test case or the failure to determine whether two failures have the sae root cause.
Here is a list of known techniques to help identify duplicated bug reports in the context of SQL test cases.

1. **Query Reduction:** Two failing SQL test cases have (likely) the same root cause if they can be reduced to the same (minimal) test case. This requires both test cases being reduced to the same minimum, which we mentioned earlier can be hard (depending on the mutation level being used). If the reducer works perfectly, all queries with the same root cause can be reduced to the same minimal reproduction test case and there are no false positives.
2. **Feature-based Clustering**: Given the test case that triggers the bug, record what features have been triggered along running the query. If the test case triggers new combination of features, it should be put into a new bucket. This work may leverage error information to extract partial features to help achieve simple implementation.
3. **Distance-based Ranking:** This technique doesn't deduplicate the test failures, but instead tries to order the test failures such that diverse, interesting test cases are highly ranked. The idea of distance-based ranking is to check what code/feature is covered by a bug-triggering test case and try to build a vector to describe each test case. Only if the vector has a long-enough distance from existing ones, we will put the test case into a new bucket for further analysis. One example for distance-based ranking is Query Taming [15].
4. **Change Bisection:** Two failing test cases have (likely) the same root cause if two queries appear the first time in the same change (e.g. git commit). This is a generic technique that doesn't need to be aware of the failure or the test case, making it simple to implement across different systems. However, it is orders of magnitudes more time and resource consuming than other techniques.
   The change found might be the one that surfaced the problem, not the one introducing it. (An idea is to combine the change diff with coverage information of the running query to decide whether the change introduced the bug or just surfaced it.) For the purpose of bug deduplication however, that is usually good enough.
   This method can have both false positives (commit introduces two different bugs) and false negatives (it's the same root cause, but one surfaces at a different commit). Prior query reduction (even best-effort) useful to remove unneeded features that (a) might not be supported in earlier versions or (b) trigger other bugs in older versions.
5. **Crash Stack Trace Matching:** For hard system crashes the stack traces can be matched to figure out if the crashes happened in the same place. Many such algorithms do already exist, like TraceSim [51] and FaST [46] that are already used for deduplicating fuzzer crashes [1]. For code generating DBMSs however, the stack trace information might be incomplete or without comparable symbols/function names.

A common practice in industry (like Google) is to combine the feature-based clustering and committing-based bisecting. The former is efficient but less effective, due to the complicated relations between bugs and triggered features, like many-to-many relations. The latter is more accurate but costly, since we need to run different code versions and set up various environments in a binary-search style to locate the first commit or unique parameter that leads to the bug.

Bisecting is also effective and in addition allows to directly assign the found bug to a developer (the owner of the change), however requires many resources and therefore is only doable for big companies.

Unfortunately, we did not find any automated tools for SQL test case deduplication, only for ranking them. The failures of most generic fuzzers are crashes, which can be deduplicated using their stack traces. In a DBMS however, many failures are not crashes but correctness issues.

4.4.20 Open Questions and Future Directions

4.4.20.1 Test case reduction

We identify the following questions of test case reduction are open and have no widely accepted solutions.

1. **Metric.** What is a good metric for the minimum test case and how hard is it to reach the global minimum? (global minimum required for test case deduplication) It is challenging to know the global minimum, so it is valuable to have a metric to evaluate how good a minimum test case is.
2. **Evaluation.** How to fairly evaluate reduction methods? There are already several reduction tools for SQL, but it's hard to compare them. Which reduction methods do they use, how minimal do the test cases get and what are their weaknesses?
3. **Generality.** Can we build a general reduction tool across database systems? Different database systems have various different SQL dialects, which affects the semantic analysis for reduction. Snowflake built a semantic-level reducer that might get open-sourced at one point. How useful will it be for other DMBSs due to dialect and semantic differences?
4. **Complexity.** Is writing (semantic-level) mutation rules a trivial task or as complex as writing a whole optimizer?

4.4.20.2 Test case deduplication

1. **Metric.** What metrics do we use for evaluating the uniqueness of bugs? Existing methods evaluate duplicated bugs by examining their input shapes and first-introduce-commits. However, these metrics are approximations, and false alarms exist. Towards to a clearly defined goal of test case deduplication and a fair evaluation method, it is expected to have more accurate metrics to evaluate the uniqueness of test cases.
2. **False Alarms.** How would false alarms affect the test case deduplication? Can we avoid false alarms? False alarms refer to either duplicated bugs that are not deduplicated or unique bugs that are deduplicated. False alarms may happen, and it is unclear about its impact on bug analysis. If false alarms have a significant impact, can we fully avoid the false alarms?
3. **Efficiency.** How to deduplicate test cases efficiently? Bisection, an existing method that deduplicates bugs, requires hundreds of thousands of builds for different commits of a database, which takes non-trivial time and computing resources. How can we deduplicate test cases within fewer resources?
4. **Compatibility with Test Oracles.** How to duplicate the test cases found by non-crash test oracles? Existing methods, such as bisection and stack trace, only work for crash bugs, which can be found by a single SQL statement. While, for the bugs that require a test oracle to check multiple SQL statements, how do we do the deduplication?

References

1 Rui Abreu, Franjo Ivančić, Filip Nikšić, Hadi Ravanbakhsh, and Ramesh Viswanathan. Reducing time-to-fix for fuzzer bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1130. IEEE, 2021.

2 Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. *CoRR*, abs/2001.02299, 2020.

3 Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*, pages 1421–1432, 2018.

4 T. Arvin. Comparison of different sql implementations. `http://troels.arvin.dk/db/rdbms` (visited: 2023-11), 2017.

5 Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *The 45th International Conference on Software Engineering (ICSE'23)*, May 2023.

6 David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 331–343, New York, NY, USA, 2017. Association for Computing Machinery.

7 Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. 2011.

8 Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *Proceedings of the 40th international conference on software engineering*, pages 433–444, 2018.

9 Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Comput. Surv.*, 56(2):31:1–31:40, 2024.

10 Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. Decomposing software verification into off-the-shelf components: An application to cegar. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 536–548, New York, NY, USA, 2022. Association for Computing Machinery.

11 Lukas Bulwahn. The new quickcheck for isabelle – random, exhaustive and symbolic testing under one roof. In *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.

12 S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, pages 146–166, 1989.

13 Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 183–198, New York, NY, USA, 2022. Association for Computing Machinery.

14 Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In *ACL2*, volume 70 of *EPTCS*, pages 4–19, 2011.

15 Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208, 2013.

16 Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, 2010.

17 Jay Corbett. Randomized testing of cloud spanner. `https://medium.com/@jcorbett_26889/randomized-testing-of-cloud-spanner-5286f1eaba75` (visited: 2023-11), 11 2023.

18 Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD Conference*, pages 2246–2258. ACM, 2022.

19 Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the Rust typechecker using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 482–493. IEEE Press, 2015.

20 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Inf. Softw. Technol.*, 46(15):1011–1025, 2004.

21 Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.

22 Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. Oracle database replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1159–1170, New York, NY, USA, 2008. Association for Computing Machinery.

23 Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. Do judge a test by its cover – combining combinatorial and property-based testing. In *ESOP*, volume 12648 of *Lecture Notes in Computer Science*, pages 264–291. Springer, 2021.

24 Google. Zetasql compliance tests. `https://github.com/google/zetasql/tree/master/zetasql/compliance` (visited: 2023-11), 11 2023.

25 Google. Zetasql reference engine. `https://github.com/google/zetasql/tree/master/zetasql/reference_impl` (visited: 2023-11), 11 2023.

26 Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 78–88, New York, NY, USA, 2012. Association for Computing Machinery.

27 Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, pages 784–795, 2016.

28 Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, Anaheim, CA, August 2023. USENIX Association.

29 Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB 2020)*, Tokyo, Japan, aug 2020.

30 Kevin E. Kline, Daniel Kline, and Brand Hunt. *SQL in a nutshell – a desktop quick reference (3. ed.)*. O'Reilly, 2008.

31 Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–111, 2019.

32 Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA):181:1–181:29, 2019.

33 Yu Liang, Song Liu, and Hong Hu. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of the 31st USENIX Security Symposium (USENIX 2022)*, Boston, MA, aug 2022.

34 Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

35 Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.

36 Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 167–182, New York, NY, USA, 2020. Association for Computing Machinery.

37 Ryan Rhodes Newton. Hermit: Deterministic linux for controlled testing and software bug-finding. `https://web.archive.org/web/20231102092943/https://developers.facebook.com/blog/post/2022/11/22/hermit-deterministic-linux-testing/`, 2022. Accessed 2023-11-02.

38 Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, 2017.

39 Christopher Olston and Benjamin Reed. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1221–1224, 2011.

40 Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

41 Jihyeok Park, Dongjun Youn, Kanguk Lee, and Sukyoung Ryu. Feature-sensitive coverage for conformance testing of programming language implementations. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

42 Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

43 Transaction Processing and Performance Council. Transaction processing and performance council. `https://tpc.org/`.

44 The Web Standards Project. The web standards compliance acid tests.

45 Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular Queries on Graph Databases. *Theory of Computing Systems*, 61(1):31–83, 2017.

46 Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes. Fast: A linear time stack trace alignment heuristic for crash report deduplication. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 549–560, 2022.

47 Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622, 1998.

48 SQLsmith. Sqlsmith. `https://github.com/anse1/sqlsmith` (visited: 2023-11), 11 2023.

49 Dominic Steinhöfel and Andreas Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery.

50 Alexander Thomson and Daniel J Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

51 Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. Tracesim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, pages 25–30, 2020.

52 Ming-Chuan Wu, Jingren Zhou, Nicolas Bruno, Yu Zhang, and Jon Fowler. Scope playback: Self-validation in the cloud. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, New York, NY, USA, 2012. Association for Computing Machinery.

53 Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, New York, NY, USA, 2018. Association for Computing Machinery.

54 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

55 Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *CCS*, pages 955–970. ACM, 2020.

56 Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

Participants

- Jinsheng Ba
National University of Singapore, SG
- Lawrence Benson
Hasso-Plattner-Institut, Universität Potsdam, DE
- Carsten Binnig
TU Darmstadt, DE
- Ankush Desai
Amazon – Cupertino, US
- Adam Dickinson
Snowflake Computing Inc. – Seattle, US
- Wensheng Dou
Chinese Academy of Sciences – Beijing, CN
- Stefania Dumbrava
ENSIIE – Paris, FR
- Moritz Eyssen
Snowflake – Berlin, DE
- Tim Fischer
Universität Tübingen, DE
- Florian Gerlinghoff
MotherDuck – Amsterdam, NL
- Torsten Grust
Universität Tübingen, DE
- Muhammad Ali Gulzar
Virginia Polytechnic Institute – Blacksburg, US
- Denis Hirn
Universität Tübingen, DE
- Hong Hu
Pennsylvania State University – University Park, US
- Zu-Ming Jiang
ETH Zürich, CH
- Marcel Kost
Salesforce – München, DE
- Burcu Kulahcioglu Ozkan
TU Delft, NL
- Federico Lorenzi
TigerBeetle – Cape Town, ZA
- Umang Mathur
National University of Singapore, SG
- Everett Maus
Google – Seattle, US
- Hannes Mühleisen
CWI – Amsterdam, NL
- Thomas Neumann
TU München – Garching, DE
- Danica Porobic
Oracle Switzerland – Zürich, CH
- Mark Raasveldt
DuckDB Labs – Amsterdam, NL
- Tilmann Rabl
Hasso-Plattner-Institut, Universität Potsdam, DE
- Manuel Rigger
National University of Singapore, SG
- Stan Rosenberg
Cockroach Labs – New York, US
- Anupam Sanghi
IBM India – Bangalore, IN
- Gambhir Sankalp
EPFL – Lausanne, CH
- Andrei Satarin
Google – Mountain View, US
- Russell Sears
Crystal DB – San Francisco, US
- Thodoris Sotiropoulos
ETH Zürich, CH
- Caleb Stanford
University of California – Davis, US
- Cheng Tan
Northeastern University – Boston, US
- Pinar Tözün
IT University of Copenhagen, DK
- Chengyu Zhang
ETH Zürich, CH