Data Debugging with Continuous Testing

Kıvanç Muşlu† Yuryi Brun‡ Alexandria Meliou

†Computer Science & Engineering University of Washington
Seattle, WA, USA 98195-2350
{kivanc@cs.washington.edu

‡School of Computer Science University of Massachusetts
Amherst, MA, USA 01003-9264
{brun, ameli}@cs.umass.edu

ABSTRACT

Today, systems rely as heavily on data as on the software that manipulates those data. Errors in these systems are incredibly costly, annually resulting in multi-billion dollar losses, and, on multiple occasions, in death. While software debugging and testing have received heavy research attention, less effort has been devoted to data debugging: discovering system errors caused by well-formed but incorrect data. In this paper, we propose continuous data testing: using otherwise-idle CPU cycles to run test queries, in the background, as a user or database administrator modifies a database. This technique notifies the user or administrator about a data bug as quickly as possible after that bug is introduced, leading to at least three benefits: (1) The bug is discovered quickly and can be fixed before it is likely to cause a problem. (2) The bug is discovered while the relevant change is fresh in the user’s or administrator’s mind, increasing the chance that the underlying cause of the bug, as opposed to only the discovered side-effect, is fixed. (3) When poor documentation or company policies contribute to bugs, discovering the bug quickly is likely to identify these contributing factors, facilitating updating documentation and policies to prevent similar bugs in the future. We describe the problem space and potential benefits of continuous data testing, our vision for the technique, challenges we encountered, and our prototype implementation for PostgreSQL. The prototype’s low overhead shows promise that continuous data testing can address the important problem of data debugging.

Categories and Subject Descriptors:
D.2.5 [Software Engineering]: Testing and Debugging
H.2.7 [Database Management]: Database Administration

General Terms: Design

Keywords: Database testing, continuous testing, data debugging

1. MOTIVATION

Today’s software systems rely heavily on data and have a profound effect on our everyday lives. Defects in these systems are common and extremely costly, having caused, for example, gas pipeline and spacecraft explosions [27, 35], loss of life [5, 21], and, at least twice, a near start of a nuclear war [18, 30, 37]. However, despite the prevalence of data errors [9, 12, 31, 32], while software-logic defects have received ample research attention, until recently (e.g., [1, 14]), detecting and correcting system errors caused by well-formed but incorrect data has received far less.

Data errors can arise in a variety of ways, including data entry errors (e.g., typographical errors and transcription errors from illegible text), measurement errors (e.g., the data source may be faulty or corrupted), and data integration errors [12]. These errors can be costly: Errors in spreadsheet data have led to million dollar losses [31, 32], and poor data quality has been estimated to cost the US economy more than $600 billion per year [9]. Data bugs have caused insurance companies to wrongly deny claims and fail to notify customers of policy changes [38]; agencies to miscalculate their budgets [10]; medical professionals to deliver incorrect medications to patients, resulting in at least 24 deaths in the US in 2003 [28]; and NASA to mistakenly ignore, via erroneous data cleaning, from 1973 until 1985, the Earth’s largest ozone hole over Antarctica [15].

In this paper, we aim to address a particular kind of data errors that are introduced into existing databases. Consider the following motivating example: Company Inc. maintains a database of its employees, their personal data, salaries, benefits, etc. Figure 1(a) shows a sample view of the data, and Figure 1(b) shows part of the documentation Company Inc. maintains to describe how the data is stored, to assist those using and maintaining the data. Company Inc. is facing tough times and negotiates a 5% reduction in the salaries of all employees. While updating the employee database, an administrator makes a mistake and instead of reducing the salary data, reduces all compensation data by 5%, which includes salary, health benefits, retirement benefits, and other forms of compensation.

After a couple of months of the mistake going unnoticed, Alonzo Church finally realizes that his paycheck stub indicates reduced benefits, retirement benefits, and other data errors. The insurance company denies Alonzo’s claims because the data indicates reduced benefits. He complains to the human resources office, who verify that Alonzo’s benefits should be higher, and ask the database administrator to fix Alonzo’s benefits data. The administrator, who has made hundreds of updates to the database since making the mistake, doesn’t think twice about the problem and updates Alonzo’s data, without realizing the underlying erroneous change that caused Alonzo’s, and other, data errors.

In a tragic turn of events, a month later, Alan Turing accidentally ingests cyanide and is hospitalized. With modern technology, the hospital quickly detects the poison and is able to save Alan’s life. Unfortunately, the insurance company denies Alan’s claims because his employer has been paying smaller premiums than what was negotiated for Alan’s policy. Alan sues Company Inc. for the damages. The mistake is finally discovered, too late to save the company.

Our example scenario, while hypothetical, is not much different from real-world scenarios caused by data errors [10, 28, 38]. Humans and applications modify data and often inadvertently introduce errors. While integrity constraints guard against predictable
Table 1: Company Inc.’s employee database table

<table>
<thead>
<tr>
<th>fname</th>
<th>mname</th>
<th>lname</th>
<th>dob</th>
<th>eid</th>
<th>compensation</th>
<th>salary</th>
<th>hbenefit</th>
<th>rbenefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alan</td>
<td>Mathison</td>
<td>Turing</td>
<td>23 Jun 1912</td>
<td>7323994</td>
<td>$240,540</td>
<td>$120,327</td>
<td>$10,922</td>
<td>$20,321</td>
</tr>
<tr>
<td>Alonzo</td>
<td>Church</td>
<td></td>
<td>14 Jun 1903</td>
<td>3420883</td>
<td>$248,141</td>
<td>$122,323</td>
<td>$11,200</td>
<td>$20,988</td>
</tr>
<tr>
<td>Tim</td>
<td>John</td>
<td>Berners-Lee</td>
<td>8 Jun 1955</td>
<td>4040404</td>
<td>$277,500</td>
<td>$145,482</td>
<td>$14,876</td>
<td>$25,800</td>
</tr>
<tr>
<td>Dennis</td>
<td>MacAlistair</td>
<td>Ritchie</td>
<td>9 Sep 1941</td>
<td>7632122</td>
<td>$202,484</td>
<td>$101,001</td>
<td>$10,002</td>
<td>$19,191</td>
</tr>
<tr>
<td>Marissa</td>
<td>Ann</td>
<td>Mayer</td>
<td>30 May 1975</td>
<td>9001009</td>
<td>$281,320</td>
<td>$150,980</td>
<td>$15,004</td>
<td>$26,112</td>
</tr>
<tr>
<td>William</td>
<td>Henry</td>
<td>Gates</td>
<td>28 Oct 1955</td>
<td>1277739</td>
<td>$320,022</td>
<td>$190,190</td>
<td>$19,555</td>
<td>$30,056</td>
</tr>
</tbody>
</table>

The employee database table contains a row for every Company Inc.’s employee. For each employee, there is a first, middle, and last name, date of birth, employee id, salary, health benefits, and retirement benefits. All employees must appear in this table, even after they retire or pass away. The employee id must be unique for each employee. The dob is stored in a “day month year” format, where month is the first three letters of the month, capitalized, and year is four digits.

Figure 1: Company Inc. keeps a database of its current and past employees. A sample view of the database (a) includes employee information (with the database’s attributes represented by the bold data descriptors). Company Inc. also maintains a description (b) of their employee database as documentation for database users, administrators, and maintainers.

2. CONTINUOUS DATA TESTING

While we cannot expect humans to avoid making mistakes altogether, the drama at Company Inc. could have been avoided if the database administrator became aware of the error soon after introducing it. Unfortunately, the error was not detected until months later, and by that time, its impact was irreversible. Our goal is not to stop errors from being introduced, but to shorten the time to detection as much as possible. Early detection will prevent errors from propagating and having practical impact, and will simplify correction as the user or administrator can more easily associate the occurrence of the error with recent updates.

We achieve this goal through continuous data testing, which continuously executes tests (black-box computations over a database) using otherwise-idle CPU cycles, alerting the user when the test results change. The test execution continues as long as the user is mak-
Continuous data testing poses several research challenges that we aim to address in our research:

**Test generation:** Generating appropriate tests for applications is challenging and has been explored extensively in the contexts of software testing (e.g., [22]) and database testing (e.g., [19, 20, 26]). Continuous data testing can rely on human-written tests, or tests generated by an automated tool or by a hybrid approach. For example, our prototype implementation uses human-written and template-generated query inputs, and generates query outputs by running the query on the unchanged database, thus creating regression tests. Critically, since tests are application dependent, they should be guided by the database workload, and should be representative of the expected database queries and capture the data semantics. Continuous data testing can directly benefit from complementary, future advances on automated data test generation.

**When to test:** While executing tests continuously, ignoring concurrent database activity and without targeting idle cycles, reduces the time before a user or administrator discovers a behavioral change caused by an update, it does not minimize the notification delay because the test executions are not prioritized based on the updates. Still, our prototype continuous data testing implementation showed this approach experienced only a 16–29% overhead (depending on the query workload) in database interactions. While reducing this overhead is important, this presents a reasonable starting point.

A more efficient continuous data testing implementation executes tests only when data updates occur. Our prototype uses database triggers to accomplish this by performing static analysis on the test queries to decide which triggers to add to which tables to trigger continuous data testing execution. Our preliminary experiments with this approach show a 7% improvement in the overhead. However, both the static analysis and incorporating more intricate database functionality remain future work. For example, in some scenarios, query results may change without data updates, e.g., when a new clustered index is added to a database.

**What to test:** A naïve continuous data testing implementation can execute every test in every iteration. However, each data change will only affect a subset of the tests; executing the tests not affected by the change wastes resources and delays the notification time. A more efficient continuous data testing implementation uses static analysis on the updates, to determine and run only the tests that could be affected by each update. This optimization to our prototype decreased the overhead by up to 40% over the naïve implementation.

Future work on test query prioritization, and even guided test query generation, can reduce the notification delay even further.

**How to test:** Test queries can be complex; executing them on large datasets may be slow and consume system resources. Future research will include incremental query computation, using the change to the data as an input to the query computation. For example, a test query that computes a sum does not need to be recomputed from scratch when a datum is updated; rather the previous sum can be adjusted based on the value of the update. Previous work on incremental view maintenance [6, 11] will guide our research. In other domains, incremental computation has been shown to greatly speed up data structures [33] and code compilation [17].

**User interface:** Reporting a test failure is not trivial. Our current continuous data testing prototype only indicates which test has failed; the user has to manually investigate whether the failure indicates an error. This can be a tedious process because the test query may be complex, and the failure can be non-descriptive and hard to analyze. Instead, descriptive failure summaries could include information on which and how many tuples are affected, and similarities between the affected tuples. Since the goal is to help users interpret failures by using explanations, in the same spirit as deriving explanations for query results using causal analysis [23], descriptive failure summaries can use causality theory to analyze the contributions of each update to a given failure.

**Debugging performance:** One of the goals of continuous data testing is to detect erroneous data quickly. However, changes in the data may not only affect the system output, but also performance. Changes to the data or an introduction of an index could make a test query slower (or faster) or could affect the execution’s memory footprint. By monitoring the tests’ performance, continuous data testing can become a useful tool for database tuning, quickly providing feedback on whether a change has a negative, positive, or no effect on performance.

3. RELATED WORK

It is possible to prevent data errors from being introduced [36], but this requires devising integrity constraints that anticipate all possible erroneous updates. Chen et al. [7] follow a more interactive approach, asking the user a challenge question to verify an update by comparing the answer with the post-update query execution. This approach is similar to tests, but requires manual effort and interferes with normal system use. In contrast, continuous data testing focuses on detecting errors, rather than preventing them, and has a minimal impact on normal execution.

Database testing research has focused on generating tests, discovering application logic errors, and debugging performance [19, 26], but not detecting data errors. Meanwhile extensive work automati-
cally generating regression tests (e.g., [22]) has neither focused on
data testing, nor query generation.
In the spreadsheet domain, data bugs can be discovered by finding
data outliers in the relative impact each datum has on formulas [1], by
detecting and analyzing data region clones [14], and by identifying
certain patterns, called smells [8]. In contrast, the continuous data
testing approach is system specific, uses tests that encode the system semantics, and, of course, applies to systems that use databases.
In writing software systems, running tests continuously and noti-
fying developers of test failures as soon as possible helps write better
code faster [29]. Reducing the notification time for compilation er-
ors cases fixing the compilation errors [17]. Continuous execution of
programs, even data-driven programs, such as spreadsheets, can
inform developers of the programs’ behavior as the programs are
being developed [13, 16]. Continuous integration and merging can
notify developers about merge conflicts quickly after they are cre-
ated [3, 4]. And speculative analysis can inform developers about errors they have not yet created, but are likely to soon [2, 25]. Likely
tests can also be predicted and executed to discover unexpected
behavioral changes [34]. An in-sync copy of the system or data can
allow for impure tests [24]. Overall, notifying developers sooner of
problems appears to make it easier to resolve those problems, which
is the primary goal of continuous data testing.

4. CONTRIBUTIONS
Continuous data testing is a novel technique for discovering sys-
tem errors caused by well-formed but incorrect data. The technique is
complementary to integrity constraints and existing efforts in
data cleaning because it targets semantic data errors. We have de-
scribed the problem space and potential benefits of continuous data
testing, outlined our vision for the technique, identified research
challenges, and discussed preliminary performance measurements based on our prototype implementation that support the feasibility
of continuous data testing. Our early research into continuous data
testing is encouraging and suggests it can be used to improve data
-intensive system quality, making continuous data testing a promising
 technique for addressing the important problem of data debugging.

5. REFERENCES
detection of collaboration conflicts. In ESEC/FSE, 2011.
We challenge you to certify your updates. In SIGMOD, 2011.
[10] B. Grady. Oakland unified makes $7.6M accounting error in
budget; asking schools not to count on it. Oakland Local, 2013.
UNECE, 2008.
clone detection and visualization in spreadsheets. In ICSE,
2013.
In AFIPS, 1969.
[18] B. Kennedy. War games: Soviets, fearing Western attack, pre-
Targeted genetic test SQL generation for the DB2 database.
In DBTest, 2012.
Wesley, 1995.
[23] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The
Complexity of Causality and Responsibility for Query An-
Speculative Analysis of Integrated Development Environment
dynamic symbolic execution for coverage criteria. In DBTest,
2011.
[27] T. C. Reed. At the Abyss: An Insider’s History of the Cold War.
[28] A. Robeznieks. Data entry is a top cause of medication errors.
[29] D. Saff and M. D. Ernst. An experimental evaluation of con-
[31] M. Sakal and L. Raković. Errors in building and using elec-
tronic tables: Financial consequences and minimisation tech-
[32] V. Samar and S. Patni. Controlling the information flow in
[33] A. Shankar and R. Bodik. DITTO: Automatic incrementaliza-
[34] G. Soares, E. Murphy-Hill, and R. Gheyi. Live feedback on
[35] A. G. Stephenson et al. Mars Climate Orbiter Mishap Investi-
[38] J. Yates. Data entry error wipes out life insurance coverage.
Chicago Tribune, 2005.