Preventing Data Errors with Continuous Testing

Kıvanç Muşlu[†] Yuriy Brun[UM] Alexandra Meliou[UM]

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

[UM]College of Information and Computer Science
University of Massachusetts
Amherst, MA, USA 01003-9264
{brun, ameli}@cs.umass.edu

ABSTRACT

Today, software systems that rely on data are ubiquitous, and ensuring the data's quality is an increasingly important challenge as data errors result in annual multi-billion dollar losses. While software debugging and testing have received heavy research attention, less effort has been devoted to data debugging: identifying system errors caused by well-formed but incorrect data. We present continuous data testing (CDT), a low-overhead, delay-free technique that quickly identifies likely data errors. CDT continuously executes domain-specific test queries; when a test fails, CDT unobtrusively warns the user or administrator. We implement CDT in the CONTEST prototype for the PostgreSQL database management system. A feasibility user study with 96 humans shows that CONTEST was extremely effective in a setting with a data entry application at guarding against data errors: With CONTEST, users corrected 98.4% of their errors, as opposed to 40.2% without, even when we injected 40% false positives into CONTEST's output. Further, when using CONTEST, users corrected data entry errors 3.2 times faster than when using state-of-the-art methods.

Categories and Subject Descriptors:
D.2.5 [Testing and Debugging]: Testing tools
H.2.0 [General]: Security, integrity, and protection
General Terms: Design, Reliability
Keywords: data debugging, data testing, continuous testing

1. INTRODUCTION

Data, and data quality, are critical to many of today's software systems. Data errors can be incredibly costly. Errors in spreadsheet data have led to million dollar losses [75, 76]. Database errors have caused insurance companies to wrongly deny claims and fail to notify customers of policy changes [86], agencies to miscalculate their budgets [31], medical professionals to deliver incorrect medications to patients, resulting in at least 24 deaths in the US in 2003 [71], and NASA to mistakenly ignore, via erroneous data cleaning, from 1973 until 1985, the Earth's largest ozone hole over Antarctica [40]. Poor data quality is estimated to cost the US economy more than $600 billion per year [21] and erroneous price data in retail databases alone cost US consumers $2.5 billion per year [23].

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA.

ACM 978-1-4503-3620-8/15/07 . . . $15.00.
http://dx.doi.org/10.1145/2771783.2771792.

Data errors can arise in a variety of ways, including during data entry (e.g., typographical errors and transcription errors from illegible text), measurement (e.g., the data source may be faulty or corrupted), and data integration [33]. Data entry errors have been found hard to detect and to significantly alter conclusions drawn on that data [3]. The ubiquity of data errors [21, 33, 75, 76] has resulted in significant research on data cleaning [13, 20, 38, 67, 69, 77] and data debugging [4, 37]. Such work, focusing on identifying likely data errors, inconsistencies, and statistical outliers can help improve the quality of a noisy or faulty data set. However, with few notable exceptions [15, 47], research has not focused on the problem of preventing the introduction of well-formed but incorrect data into a dataset that is in a sound state. The two approaches that have addressed guarding against unintentional errors caused by erroneous changes — using integrity constraints [82] to correct input errors probabilistically [47] and guarding against errors with automatically-generated certificates [15] — exhibit several shortcomings that we aim to overcome. For example, integrity constraints are difficult to use when valid data ranges span multiple orders of magnitude, can become outdated, and are hard to evolve. Our own evaluation shows that using integrity constraints significantly increases the time it takes users to correct errors. Meanwhile certificates require the user to manually verify each update, which is both tedious and delays the update from taking effect.

To address the challenge of reducing data errors caused by erroneous updates, we propose *continuous data testing* (CDT). CDT's goals are to precisely and quickly warn data-dependent application users or administrators (henceforth simply users) when changes to data likely introduce errors, without unnecessarily interfering with the users' workflow and without delaying the changes' effects. CDT's warnings are precise and timely, increasing the likelihood that the users associate the warning with the responsible data change and correct not only the data error but also the underling root cause (as opposed to some manifesting side effect) of the error.

In software development, testing is incomplete and requires significant manual effort, and yet it plays an instrumental role in improving software quality. The state-of-the-art of software system data error correction is analogous to formal verification and manual examination. While these mechanisms are also powerful in preventing software bugs, in practice, testing is indispensable. CDT brings the advantages of testing to the domain of software data errors while solving four significant research challenges: (1) To identify the intricate kinds of data errors that often slip by data type and range checks and cause the biggest software failures, CDT must encode and test data semantics. (2) CDT relies on test queries, and while some data-dependent software users may be able to write high-quality test queries, many may not. CDT can automatically generate test queries by observing the system's use of the data and generalizing

observed queries and results. (3) Delays between erroneous data changes and detection of test failures make it more difficult for users to associate the responsible change with the error, so CDT must be timely and precise. (4) Unlike software developers, users of data-dependent software may not know when to run tests and how to interpret test results, and CDT must run without disrupting normal system operation and present test failures in both unobtrusive and explanatory manner.

The key idea behind CDT, outlined in our recent vision paper [63], is to continuously run domain-specific *test queries* on the database in use by the software application. When the application updates the data, the test queries verify the semantic validity of that data. Critical to our approach is the existence of these tests. Previous research has tackled the problem of test data query generation [14, 46,49,64,65,68], and we do not focus specifically on test generation here. Tests can be mined from the history of queries the data-dependent application executes and from that application's source code, as we demonstrate in our empirical evaluation in Section 4. Further, the tangible benefits of CDT we demonstrate in this paper greatly outweigh the effort of writing test queries manually; such practice is common, and is in fact the norm in industrial software engineering, where writing tests manually makes up a major part of the development process [1].

This paper's main contributions are:

1. CDT, a technique that addresses the problem of discovering data bugs in software applications.
2. A CDT architecture and six design-level execution strategies that make CDT practical given the challenges inherent to data testing.
3. ConTest, an open-source CDT prototype built on top of the PostgreSQL database management system, available at `https://bitbucket.org/ameli/contest/`.
4. An evaluation of CDT effectiveness, via a feasibility user study with 96 humans engaging in data entry tasks. Without ConTest, users corrected 40.2% of their errors. With ConTest, the users corrected 100% of the errors, twice as fast. When we injected false positives into ConTest's warnings (making 40% of the warnings false), the users still corrected 98.4% of the errors, still nearly twice as fast. We then significantly extended a state-of-the-art technique to handle much more semantically complex constraints than today's database systems implement, and users were able to correct 100% of the errors, but more than three times slower than with ConTest.
5. An evaluation of CDT efficiency, using the TPC-H benchmark [81], showing that unoptimized CDT incurs less than 17% overhead on a typical database workload and that our optimized execution strategies reduce this overhead to as low as 1%.

Scope. The main goal of this paper is to demonstrate the feasibility of the CDT approach for detecting and preventing data errors in data-reliant systems. CDT applies to a broad range of data-reliant systems. Applying CDT to a particular system requires system-specific test queries to either be written manually or mined automatically, and a user interface for displaying potential errors and for correcting errors. While this paper discusses several approaches for automatically mining tests from system source code or execution logs, and guidelines for interface design, it is out of scope of this paper to evaluate the application of these methods to a broad range of software systems. The paper does demonstrate the effectiveness of both automatic query mining and an interface on a single system as a proof of concept.

carID	make	model	year	inventory	cost	price
121	Nissan	Versa	2014	23	$10,990	$13,199
96	Smart	fortwo Pure	2014	21	$12,490	$14,999
227	Ford	Fiesta	2014	9	$13,200	$15,799
160	Suzuki	SX4	2014	27	$13,699	$16,499
82	Chevrolet	Sonic	2014	15	$13,735	$16,499
311	KIA	Soul	2013	3	$13,300	$14,699
319	KIA	Soul	2014	22	$13,900	$16,999
286	Toyota	Yaris	2013	1	$13,980	$15,199
295	Toyota	Yaris	2014	11	$14,115	$16,999
511	Mercedes	C-Class	2014	21	$35,800	$45,999
513	Mercedes	R-Class	2014	7	$52,690	$62,899
799	Maserati	Quattroporte	2014	8	$102,500	$122,999
808	Maserati	GranTurismo	2014	12	$126,500	$149,999

Figure 1: A view of the internal database used by a car dealership's inventory software application.

The rest of this paper is organized as follows. Section 2 describes CDT on a simple example. Section 3 details the CDT design. Sections 4 and 5 evaluate CDT's effectiveness and efficiency, respectively. Section 6 discusses threats to our evaluation's validity. Section 7 places our work in the context of related research, and Section 8 summarizes our contributions and future work.

2. CDT INTUITION

This paper addresses a particular kind of data errors that are introduced with erroneous updates into an otherwise sound database used by a software system. (Our approach works even if the existing database contains errors. However, our goal is not to identify these, but rather to detect new errors that may be introduced.)

Consider the following motivating scenario: A car dealership owner decides to put the stock of cheaper cars on sale to encourage new customers to visit the showroom. The owner is confident that even if people come in for the discounted cars, many of them will buy the more expensive ones. The dealership maintains a computerized pricing system: The car data, including inventory, dealer costs, and prices are kept in a database, and each car bay in the showroom has a digital price display that loads directly from the database. The system also handles billing, monthly profit reporting, and other services. The owner, billing department, and sales personnel have a simple front-end application for interfacing with the database. Figure 1 shows a sample view of this database.

The dealership owner wishes to reduce by 30% the prices of all cars currently priced between $10K and $15K, but makes a typographical mistake and enters a wrong number into his front-end application, which, in turn, executes the SQL query:

```
UPDATE   cars SET price=0.7*price
WHERE    price BETWEEN 10000 AND 150000
```

(Note the accidental order of magnitude mistake of 150000 instead of 15000.) This incorrectly reduces by 30% the price of *all* the cars. Unfortunately, the owner does not yet realize his mistake.

Customers immediately start seeing the sale prices and coming in for a closer look. The salesmen are ecstatic as they are making more sales than ever, bringing in high commissions. The owner is happy too, seeing the flurry of activity and sales, noticing that his prediction that lowering the prices of cheap cars will lead to more sales of expensive cars is working. It's not until the end of the day, and after many sales, that the owner realizes his mistake, and the mistake's cost to his business.

Could existing technology prevent this problem? Current technology would have had a hard time catching and preventing this error. Integrity constraints [82] can guard against improper updates, but are rigid, and while they can, for example, identify errors that move the data outside of predefined ranges, they cannot detect the kind of errors the dealership owner made. Data clone detection can identify copy-and-paste data errors, which are both prevalent and important [37], but do not identify errors in databases without clear redundancy, such as the car dealership example. Outlier impact detection tools [4] also cannot detect such errors because such changes do not drastically alter the data distribution. While assertions allow for finer-grained detection, most modern database systems, including mySQL, PostgreSQL, Oracle, and Microsoft SQL server, do not implement assertions because of their unacceptable performance overhead. Finally, the dealership application could have employed certificate-based verification [15], which automatically generates challenge questions and requires the user to verify data changes. For example, when the owner attempted to reduce the price, the application could have halted the update and asked the owner to verify the expected outcome of the query:

```
SELECT  price
FROM    cars
WHERE   make='Maserati' AND model='Quattroporte'
```

If the owner entered $102,500, the application would observe that this expected value disagrees with the data after the price reduction. Unfortunately, such verification is highly manual, interferes with the owner's workflow, and delays the update execution, which causes a hardship in many real-time applications. Certificate-based verification delays not only faulty queries, but also all the proper queries, significantly affecting the application's responsiveness.

How does CDT solve the problem? We propose CDT, a testing-based approach to catching and preventing such errors. CDT uses the history of the application's execution to log queries executed on the internal database, (e.g., by the billing department or in generating monthly profit reports) to generate *test queries*. (Test queries can alternatively be written manually, as they typically are in software development.) In the dealership example, CDT extracts a test query from monthly reports that computes the hypothetical profit of selling the entire inventory. In the history of the application's execution, this profit has been between 15% and 25% of the cost of all the cars. So CDT creates the following test query, among others:

```
SELECT   'true'
FROM     (SELECT sum(inventory*(price-cost))
          AS profit FROM cars) P,
         (SELECT sum(inventory*cost)
          AS totalCost FROM cars) C
WHERE    P.profit BETWEEN 0.15*C.totalCost
                      AND 0.25*C.totalCost
```

CDT executes test queries in the background *continuously*, warning the user whenever a data change makes a test fail. By employing several optimizing execution strategies (Section 3), CDT can be made highly efficient and responsive and quickly warns the user when application operations unexpectedly affect the data. CDT does not interfere with the update execution, but instead quickly warns the user about potentially erroneous updates, allowing the user to correct the errors. This allows proper updates to execute quickly and for the database to remain responsive. Further, CDT does not interfere with the user's workflow, delivering unobtrusive warnings that the user may, using domain knowledge, deem non-critical and ignore temporarily or permanently. Finally, CDT's support for complex, domain-specific test queries allows for high-precision warnings and accurate error detection.

Challenges CDT addresses: The car dealership scenario is similar to real-world scenarios caused by data errors [31, 71, 86]. Humans and applications modify data and often inadvertently introduce errors. While integrity constraints guard against predictable erroneous updates, many careless, unintentional errors still make it through these barriers. Cleaning tools attempt to purge datasets of discrepancies before the data can be used, but many errors still go undetected and get propagated further through queries and other transformations. The car dealership scenario illustrates four data-error-detection challenges, which existing techniques *cannot* easily address.

1. Data semantics: To detect subtle but common and impactful data errors, techniques must be aware of the semantics of the data, and how the data are used. Syntactic and range-based techniques simply cannot work: If a change is intended to reduce a set of attributes by 30% but reduces a different set by 30%, without understanding the semantics, the data look valid. Further, if valid data span multiple orders of magnitude, detecting errors caused by a forgotten period during data entry (e.g., 1337 instead of 13.37) is impossible automatically (since both entries satisfy the integrity constraints) and too onerous manually. These errors neither create outliers nor break ranges. Domain-specific semantics are necessary to identify these errors.

2. Test queries: While CDT can be useful for software developers, it also targets application users and administrators who may have no experience writing tests. By analyzing the history of the queries the application has issued and the ranges of results of those queries, CDT automates test query generation. But as with software development, manually written tests, perhaps written by the application developers and shipped with the application, are highly beneficial and worth the effort.

3. Timeliness: When errors go undetected for long periods of time, they become obscure and may cause additional errors [6, 45, 73]. If the error is a side-effect of a deeper faulty change, discovering the error quickly after making the change helps identify the underlying cause, as opposed to only the discovered side-effect. For example, if the owner had noticed the price of a Toyota Yaris was too low immediately after issuing the price-reducing query, he would likely realize that other prices might be wrong as well. If, instead, he discovered the price hours later on a busy day, he may fix that one price and not realize what caused the error. Further, if errors are not fixed quickly, they may affect later decisions and complicate the fix. Finally, when other artifacts, such as poor documentation or workflow, contribute to errors, discovering the error quickly is more likely to lead to correcting those artifacts as well. CDT's optimizing execution strategies described in Section 3.3 ensure extremely fast feedback.

4. Unobtrusive and precise interface: Users of data-dependent software are less familiar with testing than software developers, and may neither know when to run tests nor how to interpret the results [48]. For example, the dealership owner is unlikely to think to run tests after reducing prices; even if he did, seeing the complex failed test query is unlikely to be useful. Research into interfaces for end-users [48] is highly relevant to this challenge. CDT must run the relevant tests automatically, and present the results unobtrusively, and in terms of the affected data and changes that may have invalidated that data, not in terms of the test queries. CDT achieves this by determining which tests can be affected by data changes, and running only those tests and only when necessary. Further, CDT highlights the changes that affected the tests, and which data are relevant, both of which are more familiar to the user than the tests.

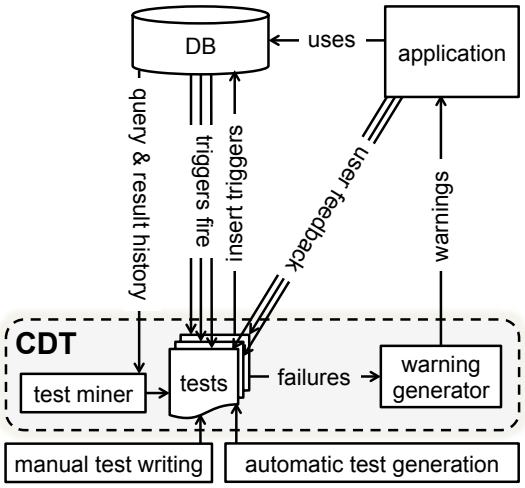


Figure 2: CDT architecture. While a software application alters data, CDT continuously runs test queries and unobtrusively warns of potentially erroneous data changes. CDT tests can be written manually, generated automatically, or mined from database use history. CDT uses database triggers to execute the tests relevant to each change and unobtrusively warns of likely erroneous data changes either via the user's application or in a separate view.

By precisely detecting semantic data errors and unobtrusively and quickly informing users of potential errors, CDT effectively tackles these four challenges.

3. CONTINUOUS DATA TESTING

CDT reduces data errors by observing executions of applications that deal with data, continuously executing tests (black-box computations over a database) in the background, and informing users with fast, targeted feedback when changes to the data are likely erroneous. CDT's goal is not to stop errors from being introduced, but to *shorten the time to detection* as much as possible. Early detection prevents errors from propagating and having practical impact, and simplifies correction as the users can more easily associate feedback with the root cause.

We developed CONTEST, a CDT prototype, based on the architecture presented in Figure 2. CDT communicates with the database via triggers that fire on data change events. When tests fail, CDT issues a warning to the application to either highlight the potential data error within the application's view or to generate a separate view of the data. The user can interact with these views to send feedback to CDT, for example indicating that a test failure is not indicative of an error. CDT then uses this information to improve its tests. When the user discovers an error, correcting the error resolves the test and removes the warning.

Our CDT design addresses four challenges described next.

3.1 Challenge 1: Data Semantics

In some domains, an acceptable data range can span multiple orders of magnitude, which thwarts general statistical outlier detection. Simultaneously, the semantics of nearly identical data can differ vastly: for example, American area codes 616 and 617 are nearly 1,000 miles apart, while 617 and 339 are geographically adjacent. In such domains, small errors difficult to detect with statistics, can significantly affect the application that uses that data.

Instead of relying on statistics, CDT uses *data semantics* to identify data errors by using domain-specific data tests. Each test consists of a query and its expected result. Tests use the data the same way applications designed for this data use it, leveraging the domain-specific data semantics to identify errors that matter in the application's domain.

Section 3.2 will describe how tests can encode data semantics, solving this challenge of identifying data errors. In particular, the most direct way to construct data tests is by mining the history of queries the application using the data in question has issued. These test queries enable encoding the semantics of the particular data and of the particular application, together with the acceptable data sensitivity.

3.2 Challenge 2: Test Query Generation

Critical to the success of CDT is the generation of high quality test queries. Test queries can be written manually, generated automatically, or mined from the history of queries and results generated by the application being tested.

Many fields and industrial practices support the manual writing of tests as one of the best ways of preventing errors [1]. We believe that the benefits of error detection that CDT offers, and the costs of errors, well outweigh the manual effort needed to write test queries. Test queries can be reused for the lifetime of the application, so the effort can be amortized. Application developers may write and ship tests with the application, but savvy users and administrators may also add tests.

Recent database advances have enabled automatically generating tests for applications that use databases [14, 46, 49, 64, 65]. CDT can use any of these approaches to generate test queries, although this paper does not evaluate these approaches' effectiveness and relative impact on CDT.

Finally, CDT can automatically mine tests from the history of queries and results in past executions of the application (and even from the source code of the application). The most commonly run queries can be generalized into tests, while test queries whose failure does not cause users to fix errors can be removed from the test suite. Further, coverage metrics can ensure test diversity. Good test candidates are commonly run queries whose aggregates, such as `MAX`, `SUM`, `AVG`, do not change significantly over time. In the car dealership example, reasonable tests include averaging all car prices, summing the total number of cars, computing the expected profit on the sales of each car, each car model, each car make, and all the cars together, as well as many other queries. If the price of a particular model remains constant for a long time, another relevant test can verify this price exactly. If the user were to change the price and receive a CDT warning, it is trivial for the user to acknowledge that, although unusual, the price has changed, and CDT can remove or update the test.

Manually written, automatically generated, and mined-from-history tests all share a desirable property: They are all highly specific to the application and the domain. Many of these tests would not be meaningful for different applications, but respect the data semantics of this particular application, making them ideal for identifying data errors and for CDT. Conveniently, the desirability of this specificity *eases* the automatic mining of test queries, reducing the effort needed to employ CDT.

3.3 Challenge 3: Timeliness

Learning about an error soon after making it increases the chances that the error is fixed, and reduces the cost of the fix [6, 45, 73]. CDT identifies potentially erroneous data changes by executing test queries, which can be computationally expensive and slow if the number of tests or the dataset are large. This can delay notifying the

user about a potential error. Further, executing tests may adversely impact the application's performance, inconveniencing the user.

To address the challenge of delivering timely feedback without adversely affecting application performance, we identify six CDT execution strategies that use both static and dynamic analysis. Section 5 will empirically evaluate these strategies and show the CDT overhead can be limited to as low as 1% in real-life workloads. Here, we first describe a naïve strategy, and then improve on it via optimizations. The optimizations do not sacrifice precision: The more efficient ones use analysis to avoid running only those tests that cannot fail, as is done by test prioritization and regression test minimization [87]. Our descriptions here are optimized for exposition, and concrete implementations of these strategies for the PostgreSQL database management system are available at `https://bitbucket.org/ameli/contest/`.

NaïveCDT: Test continuously. The most straightforward CDT implementation is to continuously run all the test queries throughout application operation. This approach is oblivious to concurrent application queries to the database, does not attempt to minimize this notification delay, and has a larger-than-necessary impact on the application performance.

SimpleCDT: Test only after changes. The first improvement is to only execute tests when application data changes. When changes are infrequent, SimpleCDT results in much lower overhead than NaïveCDT.

SmartCDT: Execute only relevant tests after relevant changes. Changes to data untouched by a test cannot change the outcome of that test, so running only those tests potentially affected by a data change can further improve performance. Using static analysis on the test queries, SmartCDT identifies which data tables and attributes the tests use, and then use database triggers to invoke the appropriate tests when the data in those tables or attributes changes.

SmartCDT$_{\mathbf{TC}}$: Test compression. Test query suites are likely to contain multiple similar tests, such as ones computing the minimum and maximum, or the sum and average of a range of data. Running such tests separately can be less efficient than a single *compressed* test. For example, the two test queries:

```
SELECT SUM(inventory)          and    SELECT AVG(cost)
FROM cars                             FROM cars
```

can be combined into a single, more efficient query:

```
SELECT SUM(inventory), AVG(cost)
FROM cars
```

SmartCDT$_{\text{TC}}$ uses dynamic analysis to compress tests: When a new test query is added to the test execution queue, SmartCDT$_{\text{TC}}$ compares it with the other tests in the queue, identifies possible compressions, and rewrites the compressed queries. This (1) reduces the time before test results are available, and (2) eliminates redundancy, which lowers the computational overhead.

SmartCDT$_{\mathbf{IT}}$: Incremental testing. Complex test queries on large datasets may be slow and may consume system resources. To address this, incremental query computation uses the *change* to the data as an input to the query computation. For example, if a single datum in a large range is changed, a test that computes a `SUM` of that range needs only to update the previously computed sum with the value of the change, rather than recompute the entire sum from scratch. SmartCDT$_{\text{IT}}$ uses dynamic analysis to identify queries in the execution queue that can be computed incrementally, uses row-level database triggers to retrieve the change values, and computes the results incrementally. Our prototype implementation (Section 5) handles queries without joins and selection predicates, but in the future, we will extend incremental testing to more complex queries using related work on incremental view maintenance [12, 32].

We refer to the version of CDT that combines all the above strategies as **SmartCDT$_{\mathbf{TC+IT}}$**.

Based on our experimental evaluation in Section 5, the above strategies are sufficient to make CDT practical. We identify three more potential execution strategies that can further increase CDT's efficiency, but leave implementing and evaluating these strategies to future work. First, in performance-critical domains, it may be worthwhile to reduce CDT overhead at the expense of increasing notification delay. Running CDT with a lower scheduling priority than the application's operations guarantees that CDT does not degrade application performance. Second, CDT can reorder tests in the execution queue to reduce the notification delay. Our implementation executes tests in FIFO priority, but could prioritize tests that are more likely to reveal an error. This prioritization can take into account prior effectiveness of each test in exposing data errors, the number of changed values that affect each test, and the time since the last execution of each test. Third, our implementation uses a coarse-grained (table- and attribute-level) static analysis to determine which data affect which tests. A finer-grained (e.g., tuple-level) and dynamic analysis that considers the concrete changed data values can better determine which changes can, and which cannot affect which tests, identifying a more precise mapping between data and tests. For example, while our static analysis will run a test that computes the `SUM` of a range whenever any datum in that range changes, a fine-grained dynamic analysis can determine that this test need not run after a change that swaps two data values in the range.

3.4 Challenge 4: Unobtrusive and Precise Interface

Because users of data-dependent software are less familiar with testing than software developers, for CDT to effectively interface with those users and identify data errors, it must (1) alleviate the user from deciding when to run the tests, (2) display test results in terms of artifacts familiar to the user, such as the data and recent changes relevant to the potential error, as opposed to the tests themselves, and (3) avoid distracting the user and altering the user's workflow.

The execution strategies described in Section 3.3 already relieve the user from worrying about when to execute the tests. CDT automatically determines which tests are relevant to run and runs them in the background.

To effectively display test failure results, CDT integrates into the application's UI and highlights data that is potentially causing an error (data touched by the failing test), and, if appropriate, the recent changes that affected this data. This makes it clear to the user that there may be a problem, where the problem likely resides, and what caused it. The user can review recent changes, decide if a change needs to be undone, and if another change needs to be enacted. The user may explicitly tell CDT that a test failure is not indicative of an error, which enables CDT to correct its expected test output.

CDT's integration into the application's UI complicates the implementation effort but is necessary to effectively interface with the user. As Figure 2 shows, when a test fails, CDT generates a warning message that contains the data and change affecting the failing test. The application then uses this warning to highlight the data and change in its own interface that is familiar to the user. Figure 3 shows an example interface of a data entry application (which we use in our user study in Section 4) highlighting three data items relevant to a failing test. For application interfaces that do not display data directly, CDT recommends creating a non-focus-stealing pane in the user interface that displays the potentially problematic

data. Highlighting can convey the severity of the error: For example, a test that computes the profit of a dealership may highlight data in yellow if the profit is within a single standard deviation of the historical median profit, orange within two standard deviations, and red outside of that range. Alternatively, the color may indicate how many failing tests are affected by a changed datum.

Finally, CDT must neither distract the user nor alter the user's workflow because obtrusive feedback that breaks the users' workflow may be detrimental to the user's productivity [7]. CDT's highlighting does not affect the application's operation, and may be hidden temporarily, again, at the discretion of the user. This makes CDT's feedback unobtrusive and gives the user the power to decide how to proceed when a warning is issued. The user may decide the error is worth investigating now, or may simply continue with the normal workflow. This makes CDT far less intrusive and distracting than prior work on detecting data errors using update certificates [15] and integrity constraints [82]. Update certificates require the user to stop current actions and provide an expected answer to an automatically generated query before proceeding, and before the data change can even be applied to the data. Integrity constraints prevent updates from taking place and requires user action before proceeding.

Our future work will include extending the user interface beyond highlighting relevant data errors and changes by using causal analysis [56], which generates human-readable explanations for test query failures by computing data's relative contributions to a query result and identifying the data most responsible for differences between test passing and test failing results.

It is important to point out that we do not envision CDT having perfect precision in practice. Good tests reduce but cannot eliminate the number of false positives — warning that arise even when the data is error-free. CDT's unobtrusive interface eases quickly inspecting and moving past the false positives, and as our user study in Section 4 shows, even when faced with 40% false positives, CDT users are able to effectively, and quickly correct errors. In contrast, techniques with obtrusive, workflow-breaking false-positive warnings are likely to annoy the users, reduce productivity, and cause the users to turn those techniques off. For example, our user study shows that integrity constraint users were more than three times as slow at correcting errors as CDT users.

4. CDT EFFECTIVENESS

We have built CONTEST, a prototype CDT implementation for the PostgreSQL database management system. The full implementation is available here: `https://bitbucket.org/ameli/contest/`. This section uses CONTEST to evaluate CDT's *effectiveness* in accomplishing its goal of reducing data errors.

The most direct way to evaluate CDT is to compare the experience of application users with and without CDT, and with prior error detecting technologies. For this evaluation, we developed a web-based, data entry application for entering ranges of data into a database and used Mechanical Turk workers as users. Figure 3 shows a screenshot of the application, whose source code is also available at `https://bitbucket.org/ameli/contest/`. The user study used real-world data and human users to most accurately evaluate CDT's effectiveness.

4.1 Real-World Data

We obtained real-world data that humans may be asked to enter in real life by extracting data from spreadsheets from `data.gov` and from the EUSES corpus [26]. We filtered the spreadsheets to select ten data tables that contained (1) at least seven columns of cells, (2) at least one column with numerical values of ten digits or more, and (3) at least one formula that used some of those numerical values (to be used as tests). From `data.gov`, we obtained seven tables from the long-range model project (`http://catalog.data.gov/dataset/long-range-model`) that contains 75-year projections of receipts, spending, deficit, and interest on US public debt. From the EUSES corpus, we obtained one financial table from FannieMae, and two tables of pulp, paper, and paperboard estimates for 1997 and 2001.

We loaded the data into a PostgreSQL relational database and generated tests in two ways: First, the formulae embedded in the spreadsheets are domain-specific uses of the data and constitute targeted data tests. Second, we use the `SUM` aggregate on the tables' columns to generate tests. In practice, these tests were sufficient to prevent false negatives.

4.2 Data Entry Application

Our web-based, data entry application first selects, at random, one row of 7–12 columns of data and creates an image depicting this row. The UI (Figure 3) displays this image, the appropriate number of cells into which to enter the data, simple instructions, an informed consent form required by the IRB, and a "Submit" button. The application submits updates to the database every time the user removes focus from a non-empty cell. For treatment groups running CDT, the application highlighted with a red background the cells used by failing tests. The application logs the values and timestamps of all data entries, and timestamps of starting the application and every submit attempt. Attempting to submit with one or more empty cells resulted in a pop up asking the user to enter all data.

4.3 Treatment Groups

Our experiment compared four treatment groups to test the effectiveness of CDT, CDT with false positives, and integrity constraints. Because the integrity constraints used by today's database management systems are highly limited (e.g., they can require data to be in a specific range, or that data values are unique, but cannot perform arbitrary computations over the data as CDT tests can), we significantly extended the integrity constraints PostgreSQL can handle to include the same semantically complex constraints as CDT tests.

Control: The control group saw no warning highlighting. The user was allowed to enter and correct data and submit once all cells were non-empty. The control group is representative of typical data entry tasks today, without the help of error-detecting tools.

CDT: The CDT group was identical to the control group, except the cells used by failing tests were highlighted in red. We used the NaïveCDT execution strategy, as performance was not a major issue in this experiment.

CDT_{FP}: The CDT_{FP} group was identical to the CDT group, except 40% of the highlighting were false positives: After the user entered *correct* data into a cell, the cell was permanently highlighted in red with a probability of 5%. (We chose to highlight 5% of the correct cells after empirically measuring the users' error rates; 5% resulted in 40% of the highlighting being false positive, and 60% true positive, as described next.)

Integrity constraints: As we have previously described, we significantly extended the PostgreSQL implementation of integrity constraints to handle the complexity of CDT queries. To emulate application use with these *extended* integrity constraints, this group was identical to the control group but clicking the "Submit" button checked all data cells, and if any mistakes were found, the application halted, informed the user there were mistakes, and returned the user to the data entry task.

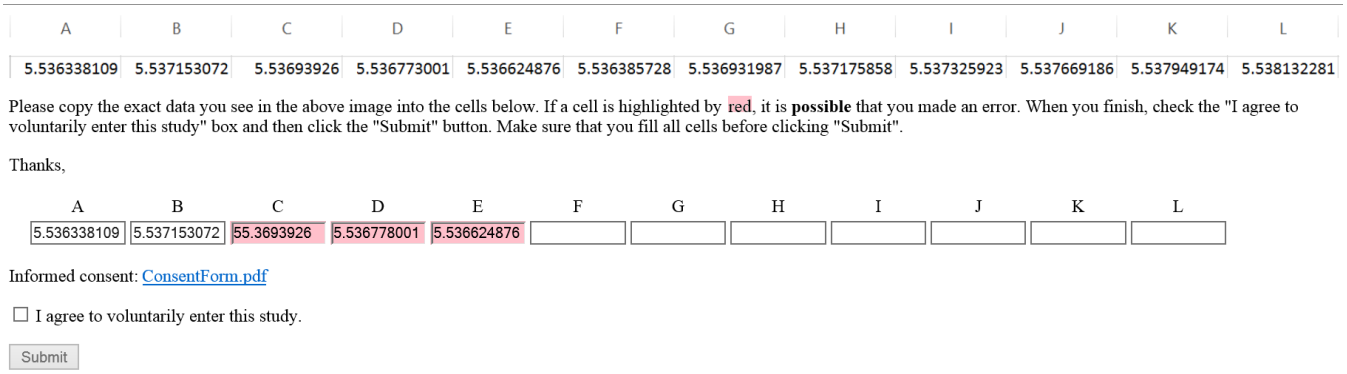


Figure 3: A screenshot of a data entry application used to evaluate CDT's effectiveness. Mechanical Turk workers were asked to enter the numbers shown at the top (displayed as images) into the cells below, and the application stored the entered data in a database. CDT warnings highlight potential errors; one of the treatment groups received false positive warnings, as is shown in cell E.

4.4 Experimental Methodology and Results

We used the Amazon Mechanical Turk (`http://www.mturk.com`) service to hire anonymous web workers to complete our data entry tasks. For each experimental group, we deployed 100 tasks, collecting a minimum of 900 cell entries per group. Overall, 96 distinct users participated in our study. We measured how often the users made erroneous entries, submitted the errors, attempted to submit, and the time it took to enter data and correct mistakes. We used a two-tailed heteroscedastic Student's t-test to analyze the statistical significance of our experimental results. Figure 4 summarizes our results.

The data entry user study answers three research questions.

> **RQ1:** How effective is CDT (with no false positives) at preventing data entry errors?

Reducing error rates is CDT's primary goal, and we first examine how well it performs in an idealized scenario with no false positives. As expected, there was no statistical significance in the data entry errors made by the users in the different configurations ($p > 0.05$), but there were differences in how the users corrected the errors. The control group users only corrected 40.2% of their errors, but CDT users corrected 100% of their errors before submitting. This result was statistically significant with $p = 0.007$.

The CDT group took 12.5% longer to complete tasks than the control group (126 sec. vs. 112 sec.), presumably because they were correcting more errors. However, the average time to correct an error was 52.2% lower for the CDT group (17.8 seconds vs. 37.2 seconds), and the difference was statistically significant ($p = 0.004$). This means that without CDT, the errors survived longer and had a higher chance of resulting in a real-world error. Extrapolating to the admittedly more complex car dealership scenario, the wrong price being in the database longer increases the chance that a car is sold at that price. Further, as time elapses after the error's introduction, fixing the underlying cause of the error becomes more difficult and costly [6, 45, 73].

> **RQ2:** Do false positives reduce CDT's effectiveness?

The CDT_{FP} group made 63 erroneous and 846 correct entries, 42 (5% of 846) of which were highlighted as false positives. This resulted in a 40% false positive rate (42 out of 105 total highlighted cells were false positive). Adding 40% false positives had minimal effect on the error submission rates — only a single error was submitted. The users were able to identify the falsely highlighted cells and still corrected 98.4% of the errors. The users were able to complete the task even faster, but this difference was not statistically significant ($p > 0.05$). Because none of our metrics showed a statistically significant difference between CDT_{FP} and CDT, we conclude that a 40% false positive rate did not impact CDT behavior.

We anticipate that higher rates of false positives will slow down the users and will impact the error correction rates as the users begin to distrust the red highlighting. Additionally, our data entry task had a simple user interface that may have mitigated the effects of false positives, and more complex tasks or tasks with less intuitive user interfaces may increase the impact of false positives. Future work will examine higher false positive rates and the effects of interfaces.

> **RQ3:** What are the effects of CDT and integrity constraints on data entry speed?

The integrity constraints group was not allowed to *submit* data with errors, so its submitted error rate was 0% (as was the CDT group's rate, without being forced). However, the CDT group was more than 3.2 times faster at correcting errors as the integrity constraint group (17.8 seconds vs. 57.3 seconds) with high statistical significance ($p = 2 \cdot 10^{-6}$). Again, correcting errors quickly reduces the chance that they propagate and cause real-world damage, reducing their cost and easing their resolution [6,45,73]. Even when faced with false positives, users were still 2.9 times faster at correcting errors (20.1 seconds vs. 57.3 seconds) with high statistical significance ($p = 3 \cdot 10^{-7}$). Overall, CDT users were 18.2% faster than those using integrity constraints (126 seconds vs. 154 seconds) at completing their entire data entry tasks with statistical significance ($p = 0.024$).

CDT (1) directs the users to the error by highlighting the relevant cells, (2) warns them about the errors sooner after the data is entered, which means the users may still be thinking about or visualizing the number they are entering, and (3) presents a sense of progress toward a goal by highlighting multiple cells when there are multiple likely errors. We expect that these factors contributed most significantly to CDT being more effective than even our extended integrity constraint mechanism, which is more powerful and flexible than what today's database management systems allow.

5. CDT EFFICIENCY

CDT must scale to real-world applications and data, and must be efficient enough to execute without significantly degrading applica-

group	total entries	errors						time	
		total		corrected		submitted		per task	to correct error
control	1,209	82	6.8%	33	40.2%	49	59.8%	112 sec.	37.2 sec.
CDT	1,097	67	6.1%	67	100 %	0	0 %	126 sec.	17.8 sec.
CDT_{FP}	909	63	6.9%	62	98.4%	1	1.6%	97 sec.	20.1 sec.
integrity constraints	1,083	50	4.6%	50	100 %	0*	0* %	154 sec.	57.3 sec.

Figure 4: Error rate and timing measurements for the data entry user study. *The integrity constraints forced a 0% error rate by disabling submission of errors.

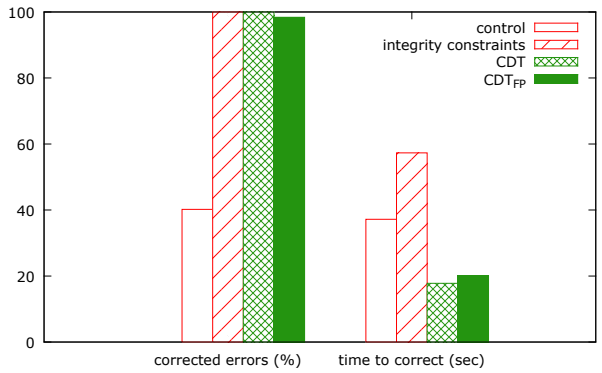


Figure 5: Using CDT, our crowd study participants identified and corrected all the errors they made, much faster than with the enforcement of integrity constraints. False positives had negligible impact both on the errors corrected and the time to correction (CDT_{FP}).

tion performance. This section evaluates ConTest's efficiency on a range of benchmark application workloads.

5.1 Benchmark Data and Workload

TPC-H [81] is a popular benchmark used in database research to simulate real-world workloads. TPC-H consists of a suite of business oriented data and queries. The data and query workloads are representative of real-world, industry-relevant situations [81]. The benchmark can be used to generate large datasets and pools of workload queries. We evaluated ConTest against a 1GB database instance and created three pools of queries for our experiments, which we later sampled to create workloads:

Read-only workload pool: TCP-H has 22 `SELECT` query templates that can be used to generate `SELECT` queries. We directly used 21 of these templates to generate our read-only workload pool. The last template involved an intermediate view, which ConTest does not currently support.

Test pool: We generated tests using 6 queries from the SELECT workload pool, omitting nested queries, which ConTest's prototype static analysis does not yet handle. We split queries with multiple aggregators into multiple test queries, one per aggregator. This created a total of 13 test queries.

Update workload pool: For each attribute in each test, we created an `UPDATE` query that changes the data for that attribute in a way that breaks the test. Thus, by design, each `UPDATE` caused at least one test to fail, and many caused multiple tests to fail.

5.2 Experimental Methodology and Metrics

Our evaluation simulated application operations using a workload generated randomly, with replacement, from the read-only workload pool. We executed this workload continuously, without pauses between queries. We randomly selected update queries from the update workload pool and injected them into this workload at regular, varying, controlled intervals, to observe how the frequency of the updates affects CDT's performance. We measured how many read-only queries were able to complete in 30 minutes as a measure of speed of the application operation. We ran the workload executions on an Intel i5 dual core 1.7 GHz processor with 4GB of RAM.

We designed these experiments as a worst-case scenario for CDT, in which the workload is executed continuously, simulating extremely demanding applications. Most of today's data-using applications are not nearly as demanding, including data entry systems and applications such as the car dealership system from Section 2. As a result, CDT overhead in those applications would be negligible. But here, we explore CDT's overhead on a performance-intensive application.

5.3 Experimental Results

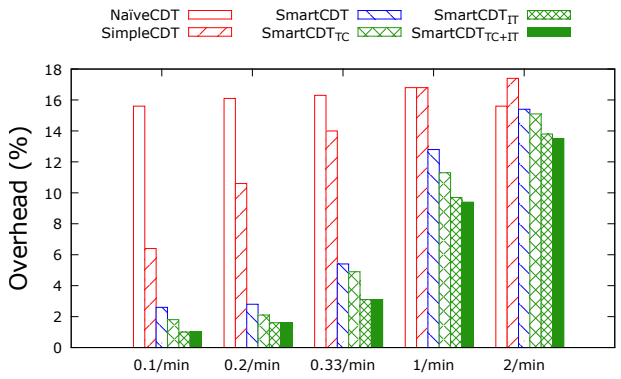
We evaluated the six CDT execution strategies described in Section 3.3. The overhead of each configuration is measured in comparison to a base execution of a 30-minute experimental run of the workload without CDT. For example, an overhead of 15.6% for NaïveCDT means that the 30-minute execution with NaïveCDT completed 15.6% fewer queries than the 30-minute execution without CDT. Figure 6 summarizes the observed overhead of the CDT executions strategies, varying the frequency of the update query workload.

> **RQ4:** What effect does CDT have on the performance of performance-intensive applications, and how well do the execution strategies mitigate this overhead?

The NaïveCDT strategy, which executes test continuously, resulted in overheads between 15.6% and 16.8%. As expected, this strategy's performance is not affected by the update frequency, since NaïveCDT is oblivious to concurrent application activity. A baseline overhead of under 17% is a great starting point for CDT and suggests that CDT is likely practical in many real-world applications.

All of our execution strategies were effective at reducing the overhead. SimpleCDT with infrequent updates (1 every 10 minutes) lowered the overhead to 6.4%, but its performance degraded quickly as the update frequency increases. SmartCDT reduced the number of tests that needed to execute, which reduced the overhead to 2.6% for infrequent updates. $SmartCDT_{TC}$ and $SmartCDT_{IT}$ further improve the overhead to 1.8% and 1.0%, respectively. $SmartCDT_{TC+IT}$ demonstrates the best performance across all workloads.

The overhead increased with the frequency of the update operations, but was never higher than 17.4%, even for these performance-intensive applications. Overall, even in this worst-case evaluation scenario, CDT performed very well and justified its use, given its effectiveness detecting errors described in Section 4.



execution strategy	overhead (per update frequency)				
	0.1/min	0.2/min	0.33/min	1/min	2/min
NaïveCDT	15.6%	16.1%	16.3%	16.8%	15.6%
SimpleCDT	6.4%	10.6%	14.0%	16.8%	17.4%
SmartCDT	2.6%	2.8%	5.4%	12.8%	15.4%
SmartCDT$_{TC}$	1.8%	2.1%	4.9%	11.3%	15.1%
SmartCDT$_{IT}$	1.0%	1.6%	3.1%	9.7%	13.8%
SmartCDT$_{TC+IT}$	1.0%	1.6%	3.1%	9.4%	13.5%

Figure 6: TPC-H benchmark performance overhead for CDT execution strategies.

6. DISCUSSION & THREATS TO VALIDITY

Test suite quality. CDT's effectiveness depends on the quality of its test queries. Our user study showed that even if tests produce 40% false positive warnings, CDT is still very effective at preventing data entry errors. This suggest that if CDT were using even less accurate tests, it would likely have a significant positive effect on maintaining data correctness, but we have not tested at what point CDT's effectiveness reduces.

Automated test generation. We have not focused in this paper on automatically generating tests for CDT. Prior work, and decades of industrial experience demonstrate that in developing software, even manually writing tests is worth the effort [1], and we believe the same is true in the data domain as well. However, such tests would likely need to be written by database administrators and system developers, not by the end users the CDT interface targets. To alleviate this concern, we have outlined several approaches for automatically mining tests from the history of application operation. We demonstrated the potential utility of these approaches in our effectiveness evaluation (Section 4) by automatically mining test queries from spreadsheet formulae. However, we have not fully evaluated the quality of tests that may be automatically mined from database systems. This threat is, however, ameliorated by the fact that database systems generally maintain a rich log of queries relevant to the applications that use them. Using queries as tests, or as templates to generate tests, is likely to lead to a powerful set of domain-specific tests, but, again, we have not yet fully evaluated this hypothesis. Even without such an evaluation, we feel that because manual test writing has been so powerful in software development, CDT provides significant value even in the unlikely scenario that data tests cannot be mined automatically effectively.

CDT user interface. An important aspect of CDT's effectiveness is its interface to notify users of potential errors and enable those users to correct those errors. This paper has discussed the challenges of this aspect of CDT and outlined guidelines for the interface design (recall Section 3.4), however, it has only built and evaluated one interface. CDT interfaces are highly system specific and would need to be designed as part of the overall system interface. Section 4 showed that even simple techniques, such as highlighting cells in data entry systems, are highly effective at preventing errors; however, data entry systems may be better suited for CDT than some other systems.

Discussion of error patterns. In our user study, we observed that CDT not only minimizes the number of errors that entry tasks introduce, but also helps users identify and fix common error patterns. Our experiments showed that several data entry errors follow specific patterns. For example, users often omitted periods and negative signs. Such errors may sometimes arise due to a user misunderstanding the task, such as assuming that periods could be omitted. By producing quick notifications of erroneous updates, CDT helps users realize this misunderstanding quickly and correct their behavior faster, saving them the work of having to retract a series of otherwise avoidable mistakes.

7. RELATED WORK

CDT combines ideas from several software engineering and database research areas.

Importance of data errors. The need for CDT is motivated by the combination of the high frequency of data errors in real-world applications, the significant impact data errors have one those real-world applications, and prevalence of end users, as opposed to trained programmers, who deal with data errors. One spreadsheet study found that 24% of spreadsheets used by the Enron Corporation contained at least one error [36]. Other studies [21, 33, 75, 76] have shown that data errors are ubiquitous. The impact of data errors is high: For example, the now famous Reinhart-Rogoff study concluded that when a nation's indebtedness crossed 90%, its growth rates plummeted [70]. This false conclusion was reached due to errors in the spreadsheet the study used [39] but had a vast impact on society as the study was used widely by politicians to justify austerity measures taken to reduce debt loads in countries around the world [39]. Meanwhile other database errors have caused insurance companies to wrongly deny claims [86], serious budget miscalculations [31], medical deaths [71], and major scientific errors [3, 40]. Erroneous price data in retail databases, as in our motivating example from Section 2, cost US consumers $2.5 billion per year [23]. Meanwhile, end users without significant programming experience who use and interpret the output of systems that rely on this erroneous data outnumber skilled programmers by an order of magnitude [2, 78]. Accordingly, CDT tackles this important problem by helping users reduce the introduction of data errors without requiring them to write programs and even tests.

Software testing. Running software tests continuously and notifying developers of test failures quickly help write better code faster [73, 74]. Reducing the notification time for compilation errors eases fixing those errors [45, 62], but can also distract and reduce productivity [7]. Continuous execution of programs, even data-driven programs such as spreadsheets, can inform developers of the programs' behavior as the programs are being developed [34, 43]. Continuous integration and merging can notify developers of merge conflicts quickly after they are created [10, 11]. These findings have led to work on simplifying the development of continuous tools for the IDE [60, 61], but not for databases. CDT focuses on preventing data errors, but the primary finding of this related work that notifying developers sooner of problems makes it easier to resolve those problems is supportive of CDT's thesis.

Data cleaning. Most data cleaning work differs from CDT by focusing on removing existing errors from large datasets, as opposed

to guarding data from new errors. Record-linkage [16, 18, 19, 51] identifies and purges integrated data that correspond to the same entity. Schema mapping [44, 59, 80] resolves data integration conflicts in schemas by generating the queries needed to transform one schema to the other. Standard approaches to cleaning include statistical outlier analysis for removing noisy data [85], interpolation to fill in missing data (e.g., with averages), and using cross-correlation with other data sources to correct or locate errors [38, 66]. Several approaches let programmers apply data cleaning tasks programmatically or interactively [28, 30, 41, 69]. Luebbers et al. [52] describe an interactive data mining approach that derives logical rules from data, and marks deviations as errors. This work is complementary to CDT, which monitors and prevents new errors.

Data bugs in spreadsheets can be discovered by finding outliers in the relative impact each datum has on formulae [4], by detecting and analyzing data region clones [37], and by identifying certain patterns, called smells [17]. By contrast, CDT uses data semantics, rather than statistics, that allow it to detect application-specific errors. CDT also applies more broadly than to only spreadsheet-based applications.

Diagnosis and explanations. Complementary to data cleaning, diagnosing techniques attempt to describe where and how errors occur. For example, Data X-Ray [83] derives optimal diagnoses as a set of features that best summarize known errors in a dataset. In a similar spirit, causality techniques [56, 57] aim to identify inputs to queries that have a large impact to a particular result. These methods focus on *explaining* errors in the data, rather than identifying, preventing, or correcting them.

Data error prevention. CDT focuses on preventing the introduction of errors. Errors introduced in data streams can be reduced using integrity constraints [47]. This approach requires users to specify integrity constraints in a special language, and probabilistically applies the constraints to incoming data to dismiss or modify erroneous data. Instead, CDT allows the user to decide how to handle errors, without requiring an a priori, one-rule-fits-all policy for handling errors. Update certificates ask the user a challenge question to verify an update by comparing the answer with the modified data [15]. Unlike CDT, this approach breaks the user's workflow and halts application execution.

Tools such as RefBook [2], BumbleBee [35], and CheckCell [4] can help prevent errors in spreadsheets. For example, RefBook and BumbleBee help programmers and end users, respectively, create refactoring macros for spreadsheet formulae, which minimizes copy-and-paste and manual editing errors. CheckCell uses the formulae encoded in a spreadsheet to identify each datum's impact and highlight the data with the most impact so that end users can prioritize verifying that data to remove high-impact data errors. Again, CDT goes beyond these techniques by using data semantics to detect application-specific errors.

Constraint-based repair. Methods that allow administrators to repair errors, typically rely on expressing these repairs as constraint optimization problems [58]. System developers can use topes to explicitly encode multiple formats for input data, which enables handling some entry errors and detecting partially correct data [77]. This process relies more heavily on the developer anticipating the kinds of data errors that may occur than CDT does, and cannot identify errors resulting from incorrect but properly formatted data in reasonable ranges. Meanwhile integrity constraints improve and maintain data quality in databases [82]. In practice, this mechanism is limited to enforcing primary (data item is unique in a range) and foreign key (inserted data item must correspond to an existing data item) constraints. Verifying integrity constraints is a blocking operation, so complex constraints are hard or impossible to enforce without significant impact on the database responsiveness, which is why most database management systems do not implement complex constraints and assertions. Our evaluation in Section 4 had to extend PostgreSQL's handling of integrity constraints. In contrast, CDT is more expressive, non-blocking, and does not interfere with the user's workflow and the application's operation. Constraint-based repair generalizes data dependencies to identify inconsistencies in the data [23, 24, 25, 29], but unlike CDT, cannot handle complex data dependencies and semantics that can be encoded with test queries.

Test generation. Database testing research has focused on generating tests [14, 79], discovering application logic errors [49], measuring the quality of existing tests [9], and debugging performance [5, 46, 64], but not detecting data errors. These techniques are complementary to CDT as they generate tests that CDT can use. In software engineering, testing and test generation work is quite extensive (e.g., [8, 22, 27, 42, 50, 53, 54, 55, 72, 84, 88]). However, unlike CDT, this work has focused neither on data testing, nor query generation.

8. CONTRIBUTIONS AND FUTURE WORK

We have presented continuous data testing (CDT), a technique for preventing the introduction of well-formed but semantically incorrect data into a dataset in use by a software application. While existing data cleaning techniques focus on identifying likely data errors, inconsistencies, and statistical outliers, they are not well suited for removing well-formatted, incorrect data that look acceptable statistically, but fail to satisfy semantic requirements.

CDT uses automatically mined and manually written test queries to test the soundness of the dataset as users and applications query and update the data. We have built CONTEST, an open-source prototype implementation of CDT available at `https://bitbucket.org/ameli/contest/`. A 96-person user study showed that CDT was extremely effective at helping users correct data errors in a data entry application, and that it does so two to three times faster than existing methods. Further, CDT is robust to poor quality tests that produce false positives. Prior work has shown that reducing the time to find an error reduces the time and cost of the fix, and prevents the error from having real-world impact [6, 45, 73]. Evaluating CDT using workloads from the TCP-H benchmark of business-oriented data and queries shows that even for performance-intensive applications, CDT's overhead is never more than 17.4%, and that our design execution strategies can reduce the overhead to as low as 1.0%.

Encouraged by the results, we foresee several interesting future directions of research. CDT can be extended to discover not only correctness errors but also performance-degrading errors, leading to database and application tuning. CDT uses static and dynamic analyses to enable its execution strategies and these analyses can be made finer-grained and more precise to further improve CDT's efficiency. CDT can be made more aware of its environment and improve its use of the available resources, such as idle CPU cycles, to improve responsiveness and efficiency. Test prioritization techniques can further reduce CDT's notification delay to improve responsiveness and effectiveness. We have discussed CDT's applicability and empirically demonstrated its potential for effectiveness. These extensions would make CDT more broadly applicable and increase its impact.

9. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants CCF-1349784, IIS-1421322, CCF-1446683, and CCF-1453508, by Google Inc. via the Faculty Research Award, and by Microsoft Research via the Software Engineering Innovation Foundation Award.

10. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 1st edition, 2008.

[2] S. Badame and D. Dig. Refactoring meets spreadsheet formulas. In *ICSM*, pages 399–409, 2012.

[3] K. A. Barchard and L. A. Pace. Preventing human error: The impact of data entry methods on data accuracy and statistical results. *Computers in Human Behavior*, 27(5):1834–1839, 2011.

[4] D. W. Barowy, D. Gochev, and E. D. Berger. CheckCell: Data debugging for spreadsheets. In *OOPSLA*, pages 507–523, 2014.

[5] A. Boehm, K.-T. Rehmann, D. H. Lee, and J. Wiemers. Continuous performance testing for SAP HANA. In *International Workshop on Reliable Data Services and Systems*, 2014.

[6] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[7] C. Boekhoudt. The big bang theory of IDEs. *Queue*, 1(7):74–82, 2003.

[8] M. Böhme and S. Paul. On the efficiency of automated testing. In *FSE*, pages 632–642, 2014.

[9] I. T. Bowman. Mutatis mutandis: Evaluating DBMS test adequacy with mutation testing. In *DBTest*, pages 10:1–10:6, 2013.

[10] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE*, pages 168–178, 2011.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, October 2013.

[12] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 577–589, 1991.

[13] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[14] D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *DBTest*, pages 6:1–6:6, 2008.

[15] S. Chen, X. L. Dong, L. V. Lakshmanan, and D. Srivastava. We challenge you to certify your updates. In *SIGMOD*, pages 481–492, 2011.

[16] A. Culotta and A. McCallum. Joint deduplication of multiple record types in relational data. In *CIKM*, pages 257–258, 2005.

[17] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva. Towards a catalog of spreadsheet smells. In *ICCSA*, pages 202–216, 2012.

[18] P. Domingos. Multi-relational record linkage. In *Workshop on Multi-Relational Data Mining*, pages 31–48, 2004.

[19] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96, 2005.

[20] X. L. Dong and F. Naumann. Data fusion: Resolving data conflicts for integration. *PVLDB*, 2(2):1654–1655, August 2009.

[21] W. W. Eckerson. Data warehousing special report: Data quality and the bottom line. `http://www.adtmag.com/article.asp?id=6321`, 2002.

[22] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, 2014.

[23] W. Fan, F. Geerts, and X. Jia. A revival of integrity constraints for data cleaning. *PVLDB*, 1(2):1522–1523, August 2008.

[24] W. Fan, F. Geerts, and X. Jia. Semandaq: A data quality system based on conditional functional dependencies. *PVLDB*, 1(2):1460–1463, August 2008.

[25] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems*, 33(2):6:1–6:48, June 2008.

[26] M. Fisher II and G. Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Software Engineering Notes*, 30(4):1–5, May 2005.

[27] J. P. Galeotti, G. Fraser, and A. Arcuri. Extending a search-based test generator with adaptive dynamic symbolic execution. In *ISSTA*, pages 421–424, 2014.

[28] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An extensible data cleaning tool. In *SIGMOD*, page 590, 2000.

[29] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, July 2013.

[30] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Data auditor: exploring data quality and semantics using pattern tableaux. *PVLDB*, 3(1-2):1641–1644, Sept. 2010.

[31] B. Grady. Oakland unified makes $7.6M accounting error in budget; asking schools not to count on it. *Oakland Local*, 2013.

[32] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[33] J. Hellerstein. Quantitative data cleaning for large databases. *UNECE*, 2008.

[34] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *ICSE*, pages 68–74, 1985.

[35] F. Hermans and D. Dig. BumbleBee: A refactoring environment for spreadsheet formulas. In *FSE Tool Demo track*, pages 747–750, 2014.

[36] F. Hermans and E. Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *ICSE SEIP track*, 2015.

[37] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen. Data clone detection and visualization in spreadsheets. In *ICSE*, pages 292–301, 2013.

[38] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.

[39] T. Herndon, M. Ash, and R. Pollin. Does high public debt consistently stifle economic growth? A critique of reinhart and rogoff. Working Paper 322, Political Economy Research Institute, University of Massachusetts Amherst, April 2013.

[40] D. Herring and M. King. Space-based observation of the Earth. *Encyclopedia of Astronomy and Astrophysics*, 2001.

[41] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *ICDE*, pages 140–142, Apr. 2006.

[42] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

[43] R. R. Karinthi and M. Weiser. Incremental re-execution of programs. In *SIGPLAN Papers of the Symposium on Interpreters and Interpretive Techniques*, 1987.

[44] V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: A context-based approach. *The VLDB Journal*, 5(4):276–304, Dec. 1996.

[45] H. Katzan Jr. Batch, conversational, and incremental compilers. In *the American Federation of Information Processing Societies*, pages 47–56, May 1969.

[46] S. A. Khalek and S. Khurshid. Systematic testing of database engines using a relational constraint solver. In *ICST*, pages 50–59, 2011.

[47] N. Khoussainova, M. Balazinska, and D. Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 43–50, 2006.

[48] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3):21:1–21:44, Apr. 2011.

[49] C. Li and C. Csallner. Dynamic symbolic database application testing. In *DBTest*, pages 7:1–7:6, 2010.

[50] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of javascript web applications. In *FSE*, pages 449–459, 2014.

[51] P. Li, C. Tziviskou, H. Wang, X. L. Dong, X. Liu, A. Maurino, and D. Srivastava. Chronos: Facilitating history discovery by linking temporal records. *PVLDB*, 5(12):2006–2009, August 2012.

[52] D. Luebbers, U. Grimmer, and M. Jarke. Systematic development of data mining-based data quality tools. In *VLDB*, pages 548–559, 2003.

[53] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of android apps. In *FSE*, pages 599–609, 2014.

[54] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Link: Exploiting the Web of data to generate test inputs. In *ISSTA*, pages 373–384, 2014.

[55] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ICASE*, page 22, 2001.

[56] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[57] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *PVLDB*, 7(13):1715–1716, 2014.

[58] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, May 2012.

[59] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[60] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Making offline analyses continuous. In *ESEC/FSE*, pages 323–333, 2013.

[61] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Reducing feedback delay of software development tools via continuous analyses. *IEEE Transactions on Software Engineering*, 2015.

[62] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *OOPSLA*, pages 669–682, 2012.

[63] K. Muşlu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *ESEC/FSE New Ideas track*, pages 631–634, 2013.

[64] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *DBTest*, pages 4:1–4:6, 2011.

[65] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *ASE*, pages 73–82, 2011.

[66] R. Pochampally, A. D. Sarma, X. L. Dong, A. Meliou, and D. Srivastava. Fusing data with correlations. In *SIGMOD*, pages 433–444, 2014.

[67] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[68] A. Raizman, A. Ananthanarayan, A. Kirilov, B. Chandramouli, and M. Ali. An extensible test framework for the Microsoft StreamInsight query processor. In *DBTest*, pages 2:1–2:6, 2010.

[69] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[70] C. M. Reinhart and K. S. Rogoff. Growth in a time of debt. *The American Economic Review*, 100(2):573–578, 2010.

[71] A. Robeznieks. Data entry is a top cause of medication errors. *American Medical News*, 2005.

[72] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE*, pages 23–32, 2011.

[73] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, 2003.

[74] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, pages 76–85, 2004.

[75] M. Sakal and L. Raković. Errors in building and using electronic tables: Financial consequences and minimisation techniques. *Strategic Management*, 17(3):29–35, 2012.

[76] V. Samar and S. Patni. Controlling the information flow in spreadsheets. *CoRR*, abs/0803.2527, 2008.

[77] C. Scaffidi, B. Myers, and M. Shaw. Topes: Reusable abstractions for validating data. In *ICSE*, pages 1–10, 2008.

[78] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[79] R. Subramonian, K. Gopalakrishna, K. Surlaker, B. Schulman, M. Gandhi, S. Topiwala, D. Zhang, and Z. Zhang. In data veritas: Data driven testing for distributed systems. In *DBTest*, pages 4:1–4:6, 2013.

[80] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. In *ICDT*, pages 182–195, 2012.

[81] Transaction Performance Council. TPC-H benchmark specification. `http://www.tpc.org/tpch`, 1999–2013.

[82] J. D. Ullman. *Principles of database and knowledge-base systems, Vol. I*. Computer Science Press, Inc., 1988.

[83] X. Wang, X. L. Dong, and A. Meliou. Data X-Ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245, 2015.

[84] Y. Wang, S. Person, S. Elbaum, and M. B. Dwyer. A framework to advise tests using tests. In *ICSE NIER track*, pages 440–443, 2014.

[85] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, March 2006.

[86] J. Yates. Data entry error wipes out life insurance coverage. *Chicago Tribune*, December 2, 2005.

[87] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.

[88] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.