Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity

Peter Bailis, Alan Fekete†, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

ABSTRACT

The rise of data-intensive "Web 2.0" Internet services has led to a range of popular new programming frameworks that collectively embody the latest incarnation of the vision of Object-Relational Mapping (ORM) systems, albeit at unprecedented scale. In this work, we empirically investigate modern ORM-backed applications' use and disuse of database concurrency control mechanisms. Specifically, we focus our study on the common use of *feral*, or application-level, mechanisms for maintaining database integrity, which, across a range of ORM systems, often take the form of declarative correctness criteria, or invariants. We quantitatively analyze the use of these mechanisms in a range of open source applications written using the Ruby on Rails ORM and find that feral invariants are the most popular means of ensuring integrity (and, by usage, are over 37 times more popular than transactions). We evaluate which of these feral invariants actually ensure integrity (by usage, up to 86.9%) and which—due to concurrency errors and lack of database support—may lead to data corruption (the remainder), which we experimentally quantify. In light of these findings, we present recommendations for database system designers for better supporting these modern ORM programming patterns, thus eliminating their adverse effects on application integrity.

1. INTRODUCTION

The rise of "Web 2.0" Internet applications delivering dynamic, highly interactive user experiences has been accompanied by a new generation of programming frameworks [80]. These frameworks simplify common tasks such as content templating and presentation, request handling, and, notably, data storage, allowing developers to focus on "agile" development of their applications. This trend embodies the most recent realization of the larger vision of object-relational mapping (ORM) systems [29], albeit at a unprecedented scale of deployment and programmer adoption.

As a lens for understanding this modern ORM behavior, we study Ruby on Rails (or, simply, "Rails") [50, 74], a central player among modern frameworks powering sites including (at one point) Twitter [35], Airbnb [9], GitHub [70], Hulu [31], Shopify [40], Groupon [67], SoundCloud [28], Twitch [71], Goodreads [2], and Zendesk [83]. From the perspective of database systems research, Rails is interesting for at least two reasons. First, it continues to be a popular means of developing responsive web application front-end and business logic, with an active open source community and user base. Rails recently celebrated its tenth anniversary and enjoys considerable commercial interest, both in terms of deployment and the availability of hosted "cloud" environments such as Heroku. Thus, Rails programmers represent a large class of consumers of database technology. Second, and perhaps more importantly, Rails is "opinionated software" [41]. That is, Rails embodies the strong personal convictions of its developer community, and, in particular, David Heinemeier Hansson (known as DHH), its creator. Rails is particularly opinionated towards the database systems that it tasks with data storage. To quote DHH:

> "I don't *want* my database to be clever! ... I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you'll have to pry that logic from my dead, cold object-oriented hands ... I want a single layer of cleverness: My domain model." [55]

Thus, this wildly successful software framework bears an actively antagonistic relationship to database management systems, echoing a familiar refrain of the "NoSQL" movement: get the database out of the way and let the application do the work.

In this paper, we examine the implications of this impedance mismatch between databases and modern ORM frameworks in the context of application integrity. By shunning decades of work on native database concurrency control solutions, Rails has developed a set of primitives for handling application integrity in the application tier—building, from the underlying database system's perspective, a *feral* concurrency control system. We examine the design and use of these feral mechanisms and evaluate their effectiveness in practice by analyzing them and experimentally quantifying data integrity violations in practice. Our goal is to understand how this growing class of applications currently interacts with database systems and how we, as a database systems community, can positively engage with these criticisms to better serve the needs of these developers.

We begin by surveying the state of Rails' application-tier concurrency control primitives and examining their use in 67 open source applications representing a variety of use cases from e-Commerce to Customer Relationship Management and social networking. We find that, these applications overwhelmingly use Rails' built-in support for declarative invariants—*validations* and *associations*—to protect data integrity—instead of application-defined transactions, which are used more than 37 times less frequently. Across the survey, we find over 9950 uses of application-level validations designed to ensure correctness criteria including referential integrity, uniqueness, and adherence to common data formats.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2737784.

Given this corpus, we subsequently ask: are these feral invariants correctly enforced? Do they work in practice? Rails will execute validation checks concurrently, so we study the potential for data corruption due to races if validation and update activity does not run within a serializable transaction in the database. This is a real concern, as many DBMS platforms use non-serializable isolation by default and in many cases (despite labeling otherwise) do not provide serializable isolation as an option at all. Accordingly, we apply invariant confluence analysis [17] and show that, in fact, up to 86.9% of Rails validation usage by volume is actually safe under concurrent execution. However, the remainder—which include uniqueness violations under insertion and foreign key constraint violations under deletion—are not. Therefore, we quantify the impact of concurrency on data corruption for Rails uniqueness and foreign key constraints under both worst-case analysis and via actual Rails deployment. We demonstrate that, for pathological workloads, validations reduce the severity of data corruption by orders of magnitude but nevertheless still permit serious integrity violations.

Given these results, we return to our goal of improving the underlying data management systems that power these applications and present recommendations for the database research community. We expand our study to survey several additional web frameworks and demonstrate that many also provide a notion of feral validations, suggesting an industry-wide trend. While the success of Rails and its ilk—despite (or perhaps due to) their aversion to database technology—are firm evidence of the continued impedance mismatch between object-oriented programming and the relational model, we see considerable opportunity in improving database systems to better serve these communities—via more programmer- and ORM-friendly interfaces that ensure correctness while minimizing impacts on performance and portability.

In summary, this paper makes the following contributions:

- We analyze 67 open source Ruby on Rails applications to determine their use of both database-backed and feral concurrency control mechanisms. This provides a quantitative picture of how mainstream web developers interact with database systems, and, more specifically, concurrency control.
- We study these applications' feral mechanisms potential for application integrity violations. We analytically and experimentally quantify the incidence and degree of inconsistency allowed by Rails's uniqueness and association validations.
- We survey six additional frameworks for similarly unsafe validations. Based on these results and those above, we present a set of recommendations for database systems designers, including increasing database support for application invariants while avoiding coordination and maintaining portability.

In all, this paper is an attempt to understand how a large and growing class of programmers and framework authors interacts with the data management systems that this community builds. We hope to raise awareness about prevalent and under-supported application programming patterns and their impact on the integrity of real-world, end-user database-backed applications. Our contributions do not include a new system for database concurrency control; rather, our goal is to inform designers and architects of next-generation data management systems and provide quantitative evidence of the practical shortcomings and pitfalls in real-world database concurrency control today. We view this work as an early example of the promising opportunity in empirical analysis of open source database-backed software as written and deployed in practice.

The remainder of this paper proceeds as follows. Section 2 briefly provides background on Rails MVC and deployment, while Section 3 surveys Rails's supported concurrency control mechanisms. Section 4 presents analysis of mechanism usage in open source applications as well as safety under weak isolation. Section 5 experimentally quantifies the integrity violations allowed in a Rails deployment. Section 6 describes support for feral validations in additional frameworks, and Section 7 presents recommendations for better supporting these framework demands. Section 8 presents related work, and Section 9 concludes with a reflection on the potential of empirical methods in database systems research.

2. BACKGROUND

As a primary focus of our study, we investigate the operational model, database use, and application primitives provided in Rails. In this section, we provide a brief overview of the Rails programming model and describe standard Rails deployment architectures.

2.1 Rails Tenets and MVC

Rails was developed in order to maximize developer productivity. This focus is captured by two core architectural principles [74]. First, Rails adopts a "Don't Repeat Yourself" (DRY) philosophy: "every piece of knowledge should be expressed in just one place" in the code. Data modeling and schema descriptions are relegated to one portion of the system, while presentation and business logic are relegated to two others. Rails attempts to minimize the amount of boilerplate code required to achieve this functionality. Second, Rails adopts a philosophy of "Convention over Configuration," aiming for sensible defaults and allowing easy deployment without many—if any—modifications to configuration.

A natural corollary to the above principles is that Rails encourages an idiomatic style of programming. The Rails framework authors claim that "somehow, [this style] just seems right" for quickly building responsive web applications [74]. The framework's success hints that its idioms are, in fact, natural to web developers.

More concretely, Rails divides application code into a three-component architecture called Model-View-Controller [49, 60]:

- The **Model** acts as a basic ORM and is responsible for managing business objects, including schemas, querying, and persistence functionality. For example, in a banking application, an account's state could be represented by a model with a numeric owner ID field and a numeric balance field.
- The **View** acts as a presentation layer for application objects, including rendering into browser-ingestible HTML and/or other formats such as JSON. In our banking application, the View would be responsible for rendering the page displaying a user's account balance.
- The **Controller** encapsulates the remainder of the application's business logic, including actual generation of queries and transformations on the Active Record models. In our banking application, we would write logic for orchestrating withdrawal and deposit operations within the Controller.

Actually building a Rails application is a matter of instantiating a collection of models and writing appropriate controller and presentation logic for each.

As we are concerned with how Rails utilizes database backends, we largely focus on how Rails applications interact with the Model layer. Rails natively supports a Model implementation called `Active Record`. Rails's Active Record module is an implementation of the Active Record pattern originally proposed by Martin Fowler, a prominent software design consultant [48]. Per Fowler, an Active Record is "an object that wraps a row in a database or view, encapsulates the database access, and adds domain logic on that data" (further references to Active Record will correspond to Rails's

implementation). The first two tasks—persistence and database encapsulation—fit squarely in the realm of standard ORM design, and Rails adopts Fowler's recommendation of a one-to-one correlation between object fields and database columns (thus, each declared Active Record class is stored in a separate table in the database). The third component, domain logic, is more complicated. Each Rails model may contain a number of attributes (and must include a special primary-key-backed `id` field) as well as associated logic including data validation, associations, and other constraints. Fowler suggests that "domain logic that isn't too complex" is well-suited for encapsulation in an Active Record class. We will discuss these in greater depth in the next section.

2.2 Databases and Deployment

This otherwise benign separation of data and logic becomes interesting when we consider how Rails servers process concurrent requests. In this section, we describe how, in standard Rails deployments, application logic may be executed concurrently and without synchronization within separate threads or processes.

In Rails, the database is—at least for basic usages—simply a place to store model state and is otherwise divorced from the application logic. All application code is run within the Ruby virtual machine (VM), and Active Record makes appropriate calls to the database in order to materialize collections of models in the VM memory as needed (as well as to persist model state). However, from the database's perspective (and per DHH's passionate declaration in Section 1), logic remains in the application layer. Active Record natively provides support for PostgreSQL, MySQL, and SQLite, with extensions for databases including Oracle and is otherwise agnostic to database choice.

Rails deployments typically resemble traditional multi-tier web architectures [10] and consist of an HTTP server such as Apache or Nginx that acts as a proxy for a pool of Ruby VMs running the Rails application stack. Depending on the Ruby VM and Rails implementation, the Rails application may or may not be multi-threaded.[1] Thus, when an end-user makes a HTTP request on a Rails-powered web site, the request is first accepted by a web server and passed to a Rails worker process (or thread within the process). Based on the HTTP headers and destination, Rails subsequently determines the appropriate Controller logic and runs it, including any database calls via Active Record, and renders a response via the View, which is returned to the HTTP server.

Thus, in a Rails application, the *only* coordination between individual application requests occurs within the database system. Controller execution—whether in separate threads or across Ruby VMs (which may be active on different physical servers)—is entirely independent, save for the rendezvous of queries and modifications within the database tier, as triggered by Active Record operations.

The independent execution of concurrent business logic should give serious pause to disciples of transaction processing systems. Is this execution strategy actually safe? Thus far, we have yet to discuss any mechanisms for maintaining correct application data, such as the use of transactions. In fact, as we will discuss in the next section, Rails has, over its lifetime, introduced several mechanisms for maintaining consistency of application data. In keeping with Rails' focus on keeping application logic within Rails (and not in the database), this has led to several different proposals. In the remainder of this paper, we examine their use and whether, in fact, they correctly maintain application data.

[1] Ruby was not traditionally designed for highly concurrent operations: its standard reference VM—Ruby MRI—contains (like Python's CPython) a "Global VM Lock" that prevents multiple OS threads from executing at a given time. While alternative VM implementations provide more concurrent behavior, until Rails 2.2 (released in November 2008), Rails embraced this behavior and was unable to process more than one request at a time (due to state shared state including database connections and logging state) [69]. In practice today, the choice of multi-process, multi-threaded, or multi-process and multi-threaded deployment depends on the actual application server architecture. For example, three popular servers—Phusion Passenger, Puma, and Unicorn —each provide a different configuration.

3. FERAL MECHANISMS IN RAILS

As we discussed in Section 2.2, Rails services user requests independently, with the database acting as a point of rendezvous for concurrent operations. Given Rails's design goals of maintaining application logic at the user level, this appears—on its face—a somewhat cavalier proposition with respect to application integrity. In response, Rails has developed a range of concurrency control strategies, two of which operate external to the database, at the application level, which we term *feral concurrency control* mechanisms.

In this section, we outline four major mechanisms for guarding against integrity violations under concurrent execution in Rails. We subsequently begin our study of 67 open source applications to determine which of these mechanisms are used in practice. In the following section, we will determine which are sufficient to maintain correct data—and when they are not.

3.1 Rails Concurrency Control Mechanisms

Rails contains four main mechanisms for concurrency control.

1. Rails provides support for **transactions**. By wrapping a sequence of operations within a special `transaction` block, Rails operations will execute transactionally, backed by an actual database transaction. The database transaction either runs at the database's configured default isolation level or, as of Rails 4.0.0, can be configured on a per-transaction basis [64].
2. Rails provides support for both optimistic and pessimistic per-record **locking**. Applications invoke pessimistic locks on an Active Record object by calling its `lock` method, which invokes a `SELECT FOR UPDATE` statement in the database. Optimistic locking is invoked by declaring a special `lock_version` field in an Active Record model. When a Rails process performs an update to an optimistically locked model, Active Record atomically checks whether the corresponding record's `lock_version` field has changed since the process last read the object; if it has not changed, Rails transactionally increments `lock_version` and updates the database record.
3. Rails provides support for application-level **validations**. Each Active Record model has a set of zero or more validations, or boolean-valued functions, and a model instance many only be saved to the database if all of its declared validations return true. These validations ensure, for example, that particular fields within a record are not null or, alternatively, are unique within the database. Rails provides a number of built-in validations but also allows arbitrary user-defined validations (we discuss actual validations further in subsequent sections). The framework runs each declared validation sequentially and, if all succeed, the model state is updated in the database; this happens within a database-backed transaction.[2] The validations supported by Rails today include ones that are natively supported by many commercial databases today, as well as others.

[2] The practice of wrapping validations in a transaction dates to the earliest public Rails commit (albeit, in 2004, transactions were only supported via a per-Ruby VM global lock [54]). However, as late as 2010, updates were only partially protected by transactions [75].

4. Rails provides support for application-level **associations**. As the name suggests, "an association is a connection between two Active Record models," effectively acting like a foreign key in an RDBMS. Associations can be declared on one or both sides of a one-to-one or one-to-many relationship, including transitive dependencies (via a `:through` annotation). Declaring an association (e.g., `:belongs_to dept`) produces a special field for the associated record ID within the model (e.g., `dept_id`). Coupling an association with an appropriate validation (e.g., `:presence`) ensures that the association is indeed valid (and is, via the validation, backed by a database transaction). Until the release of Rails 4.2 in December 2014, Rails did not provide native support for database-backed foreign key constraints. In Rails 4.2, foreign keys are supported via manual schema annotations declared separately from each model; declaring an association does not declare a corresponding foreign key constraint and vice-versa.

Overall, these four mechanisms provide a range of options for developers. The first is squarely in the realm of traditional concurrency control. The second is, in effect, a coarse-grained user-level implementation of single-record transactions via database-level "compare-and-swap" primitives (implemented via `SELECT FOR UPDATE`). However, the latter two—validations and associations—operate, in effect, at the application level. Although some validations like uniqueness validations have analogs in an RDBMS, the semantics of these validations are entirely contained within the Rails code. In effect, from the database's perspective, these validations exist external to the system and are *feral* concurrency control mechanisms.

Rails's feral mechanisms—validations and associations—are a prominent feature of the Active Record model. In contrast, neither transactions nor locks are actually discussed in the official "Rails Guides," and, generally, are not promoted as a means of ensuring data integrity. Instead, the Rails documentation [7] prefers validations as they are "are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain." Moreover, the Rails documentation opines that "database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult." As we will show shortly, these feral mechanisms accordingly dominate in terms of developer popularity in real applications.

3.2 Adoption in Practice

To understand exactly how users interact with these concurrency control mechanisms and determine which deserved more study, we examined their usage in a portfolio of publicly available open source applications. We find that validations and associations are overwhelmingly the most popular forms of concurrency control.

Application corpus. We selected 67 open source applications built using Ruby on Rails and Active Record, representing a variety of application domains, including eCommerce, customer relationship management, retail point of sale, conference management, content management, build management, project management, personal task tracking, community management and forums, commenting, calendaring, file sharing, Git hosting, link aggregation, crowdfunding, social networking, and blogging. We sought projects with substantial code-bases (average: 26,809 lines of Ruby) multiple contributors (average: 69.1), and relative popularity (measured according to GitHub stars) on the site. Table 2 (in the Appendix) provides a detailed overview.

While several of these applications are projects undertaken by hobbyists, many are either commercially supported (e.g., Canvas LMS, Discourse, Spree, GitLab) and/or have a large open source community (e.g., Radiant, Comfortable Mexican Sofa, Diaspora). A larger-scale commercial, closed-source Rails application such as Twitter, GitHub, or Airbnb might exhibit different trends than those we observe here. However, in the open source domain, we believe these applications represent a diverse selection of Rails use cases and are a good-faith effort to obtain a representative sample of popular open source Rails applications as hosted on GitHub.

Mechanism usage. We performed a simple analysis of the applications to determine how each of the concurrency control mechanisms were used (see Appendix A for more methodological details).

Overwhelmingly, applications did not use transactions or locks (Figure 1 and Table 2). On average, applications used 0.13 transactions, 0.01 locks, 1.80 validations, and 3.19 associations per model (with an average of 29.1 models per application). While 46 (68.7%) of applications used transactions, all used some validations or associations. Only six applications used locks. Use of pessimistic locks was over twice as common as the use of optimistic locks.

Perhaps most notable among these general trends, we find that validations and associations are, respectively, 13.6 and 24.2 times more common than transactions and orders of magnitude more common than locking. These feral mechanisms are—in keeping with the Rails philosophy—favored by these application developers. That is, rather than adopting the use of traditional transactional programming primitives, Rails application writers chose to instead specify correctness criteria and have the ORM system enforce the criteria on their behalf. It is unclear and even unlikely that these declarative criteria are a complete specification of program correctness: undoubtedly, some of these programs contain errors. However, given that these criteria are nevertheless being declared by application writers and represent a departure from traditional, transaction-oriented programming, we devote much of the remainder of this work to examining exactly what they are attempting to preserve (and whether they are actually sufficient to do so).

Understanding specific applications. Over the course of our investigation, we found that application use of mechanisms varied. While our focus is largely on aggregate behavior, studying individual applications is also interesting. For example, consider Spree, a popular eCommerce application:

Spree uses only six transactions, one for each of 1.) canceling an order, 2.) approving an order (atomically setting the user ID and timestamp), 3.) transferring shipments between fulfillment locations (e.g., warehouses), 4.) transferring items between shipments, 5.) transferring stock between fulfillment locations, and 6.) updating an order's specific inventory status. While this is a reasonable set of locations for transactions, in an eCommerce application, one might expect a larger number of scenarios to require transactions, including order placement and stock adjustment.

In the case of Spree stock adjustment, the inventory count for each item is a potential hotspot for concurrency issues. Manual adjustments of available stock (`adjust_count_on_hand(value)`) is indeed protected via a pessimistic lock, but simply setting the available stock (`set_count_on_hand(value)`) is not. It is unclear why one operation necessitates a lock but the other does not, given that both are ostensibly sensitive to concurrent accesses. Meanwhile, the stock level field is wrapped in a validation ensuring non-negative balances, preventing negative balances but not necessarily classic Lost Update anomalies [8].

At one point, Spree's inventory count was protected by an optimistic lock; it was removed due to optimistic lock failure during customer checkouts. On relevant GitHub issue pertaining to this lock removal, a committer notes that "I think we should get rid of the [optimistic lock] if there's no documentation about why it's there...I think we can look at this issue again in a month's time

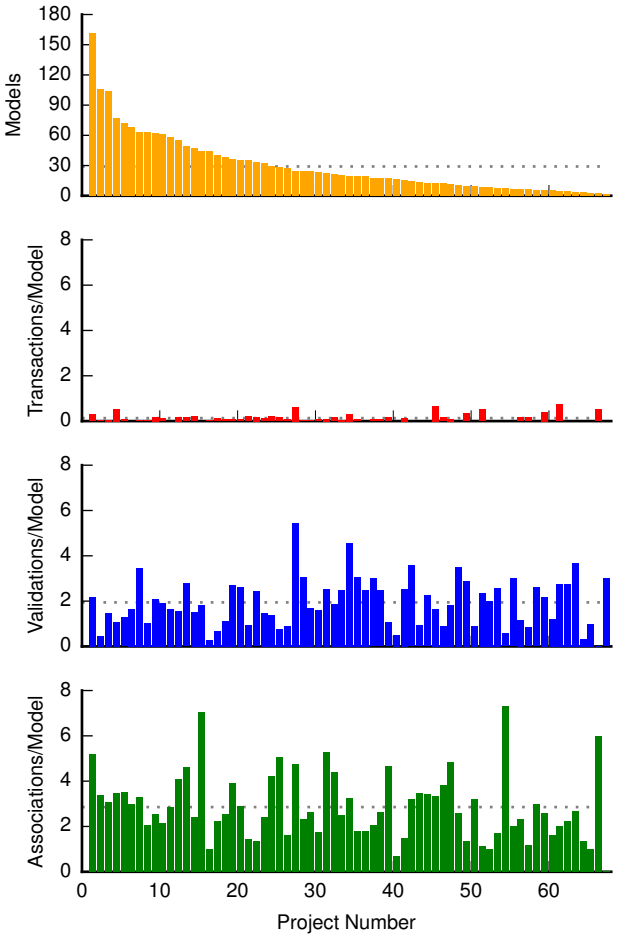

Figure 1: Use of concurrency control mechanisms in Rails applications. We maintain the same ordering of applications for each plot (i.e., same x-axis values; identical to Table 2) and show the average for each plot using the dotted line.

and see if there's been any problems since you turned it off" [38]. This removal has, to our knowledge, not been revisited, despite the potential dangers of removing this point of synchronization.

The remainder of the application corpus contains a number of such fascinating examples, illustrating the often ad-hoc process of deciding upon a concurrency control mechanism. Broadly, the use of each style of concurrency control varies across repositories, but our results demonstrate a clear trend towards feral mechanisms within Rails rather than traditional use of transactions.

Additional metrics. To better understand how programmers used each of these mechanisms, we performed two additional analyses.

First, we analyzed the number of models, transactions, validations, and associations over each project's lifetime. Using each project's Git history, we repeated the above analysis at a fixed set of intervals through the project's lifespan (measured by commits). Figure 6 (see Appendix) plots the median number of occurrences across all projects. The results show that concurrency control mechanisms (of all forms) tend to be introduced after models are introduced. That is, additions to the data model precede (often by a considerable amount) additional uses of transactions, validations, and associations. It is unclear whether the bulk of concurrency control usage additions are intended to correct concurrency issues or are instead due to natural growth in Controller code and business logic. However, the gap between models and concurrency control usage shrinks over time; thus, the data model appears to stabilize faster than the controller logic, but both eventually stabilize. We view additional longitudinal analysis along these lines as worthwhile future work.

Second, we analyze the distribution of authors to commits compared to the distribution of authors to validations and associations authored.[3] As Figure 7 (see Appendix, page) demonstrates, 95% of all commits are authored by 42.4% of authors. However, 95% of invariants (validations plus associations) are authored by only 20.3% of authors. This is reminiscent of traditional database schema authorship, where a smaller number of authors (e.g., DBAs) modify the schema than contribute to the actual application code.

3.3 Summary and Discussion

Returning to the Rails design philosophy, the applications we have encountered do indeed express their logic at the application layer. There is little actual communication of correctness criteria to the database layer. Part of this is due to limitations within Rails. As we have mentioned, there is no way to actually declare a foreign key constraint in Rails without importing additional third-party modules. Insofar as Rails is an "opinionated" framework encouraging an idiomatic programming style, if our application corpus is any indication, DHH and his co-authors advocating application-level data management appear to have succeeded en masse.

Having observed the relative popularity of these mechanisms, we turn our attention to the question of their correctness. Specifically, do these application-level criteria actually enforce the constraints that they claim to enforce? We restrict ourself to studying declared validations and associations for three reasons. First, as we have seen, these constructs are more widely used in the codebases we have studied. Second, these constructs represent a deviation from standard concurrency control techniques and are therefore perhaps more likely to contain errors. Third, while analyzing latent constraints (e.g., those that might be determined via more sophisticated techniques such as pre- and post-condition invariant mining [65, 73] and/or by interviewing each developer on each project) would be instructive, this is difficult to scale. We view these forms of analysis as highly promising avenues for future research.

4. ISOLATION AND INTEGRITY

We now turn our attention to understanding which of Rails' feral validations and associations are actually correct under concurrent execution as described in Section 2.2 and which require stronger forms of isolation or synchronization for correct enforcement.

4.1 Understanding Validation Behavior

To begin, recall that each sequence of validations (and model update as well, if validations pass) is wrapped within a database-backed transaction, the validation's intended integrity will be preserved provided the database is using serializable isolation. However, relational database engines often default to non-serializable isolation [16]; notably for Rails, PostgreSQL and MySQL actually default to, respectively, the weaker Read Committed and Repeatable Read isolation levels.

We did not encounter evidence that applications changed the isolation level. Rails does not configure the database isolation level for validations, and none of the application code or configurations we encountered change the default isolation level, either (or mention doing so in documentation). Thus, although we cannot prove that

[3] We chose to analyze commits authored rather than lines of code written because git tracks large-scale code refactoring commits as an often large set of deletions and insertions. Nevertheless, we observed a close correlation between lines of code and commits authored.

this is indeed the case, this data suggests that validations are likely to run at default database isolation in production environments.

Validations with weak isolation. Given that validations are not likely to be perfectly isolated, does this lack of serializable isolation actually affect these invariants? Just because validations effectively run concurrently does not mean that they are necessarily incorrect. To determine exactly which of these invariants are correct under concurrent execution, we draw on the recently developed theory of invariant confluence [17].

Invariant confluence (I-confluence) provides a necessary and sufficient condition for whether or not invariants can be preserved under coordination-free, concurrent execution of transactions. Informally, the condition ensures that, if transactions maintain database states that are correct with respect to an invariant when run in isolation, the transactions can be run simultaneously and their results combined ("merged") to produce another correct state. In the case of Rails, we wish to determine whether, in the event of concurrent validations and model saves, the result of concurrent model saves will not violate the validation for either model. In the event that two concurrent controllers save the same model (backed by the same database record), only one will be persisted (a some-write-wins "merge"). In the event that two concurrent controllers save different models (i.e., backed by different database records), both will be persisted (a set-based "merge"). In both cases, we must ensure that validations hold after merge.

Our I-confluence analysis currently relies on a combination of manual proofs and simple static analysis: given a set of invariant and operation pairs classified as providing the I-confluence property, we can iterate through all operations and declared invariants and check whether or not they appear in the set of I-confluent pairs. If so, we label the pair as I-confluent. If not, we can either conservatively label the pair as unsafe under concurrent execution or prove the pair as I-confluent or not. (To prove a pair is I-confluent, we must show that the set of database states reachable by executing operations preserves the invariant under merge, as described above.)

Returning to our task of classifying Rails validations and associations as safe or not, we applied this I-confluence analysis to the invariants[4] in the corpus. In our analysis, we found that only 60 out of 3505 validations were expressed as user-defined functions. The remainder were drawn from the standard set of validations supported by Rails core.[5] Accordingly, we begin by considering built-in validations, then examine each of the custom validations.

4.2 Built-In Validations

We now discuss common, built-in validations and their I-confluence. Many are I-confluent and are therefore safe to execute concurrently.

Table 1 presents the ten most common built-in validations by usage and their occurences in our application corpus. The exact coordination requirements depended on their usage.

The most popular invariant, `presence`, serves multiple purposes. Its basic behavior is to simply check for empty values in a model before saving. This is I-confluent as, in our model, concurrent model saves cannot result in non-null values suddenly becoming null. However, `presence` can also be used to enforce that the opposite end of an association is, in fact, present in the database (i.e., referential integrity). Under insertions, foreign key constraints are I-confluent [17], but, under deletions, they are not.

[4] We focus on validations here as, while associations *do* represent an invariant, it is only when they are coupled with validations that they are enforced.

[5] It is unclear exactly why this is the case. It is possible that, because these invariants are standardized, they are more accessible to users. It is also possible that Rails developers have simply done a good job of codifying common patterns that programmers tend to use.

Name	Occurrences	I-Confluent?
`validates_presence_of`	1762	Depends
`validates_uniqueness_of`	440	No
`validates_length_of`	438	Yes
`validates_inclusion_of`	201	Yes
`validates_numericality_of`	133	Yes
`validates_associated`	39	Depends
`validates_email`	34	Yes
`validates_attachment_content_type`	29	Yes
`validates_attachment_size`	29	Yes
`validates_confirmation_of`	19	Yes
Other	321	

Table 1: Use of and invariant confluence of built-in validations.

The second most popular invariant, concerning record uniqueness, is *not* I-confluent [17]. That is, if two users concurrently insert or modify records, they can introduce duplicates.

Eight of the next nine invariants are largely concerned with data formatting and are I-confluent. For example, `numericality` ensures that the field contains a number rather than an alphanumeric string. These invariants are indeed I-confluent under concurrent update.

Finally, the safety of `associated` (like `presence`) is contingent on whether or not the current updates are both insertions (I-confluent) or mixed insertions and deletions (not I-confluent). Thus, correctness depends on the operation.

Overall, a large number of built-in validations are safe under concurrent operation. Under insertions, 86.9% of built-in validation occurrences as I-confluent. Under deletions, only 36.6% of occurrences are I-confluent. However, associations and multi-record uniqueness are—depending on the workload—not I-confluent and are therefore likely to cause problems. In the next section, we examine these validations in greater detail.

4.3 Custom Validations

We also manually inspected the coordination requirements of the 60 (1.71%) validations (from 17 projects) that were declared as UDFs. 52 of these custom validations were declared inline via Rails's `validates_each` syntax, while 8 were custom classes that implemented Rails's validation interface. 42 of 60 validations were I-confluent, while the remaining 18 were not. Due to space constraints, we omit a discussion of each validation but discuss several trends and notable examples of custom validations below.

Among the custom validations that were I-confluent, many consisted of simple format checks or other domain-specific validations, including credit card formatting and static username blacklisting.

The validations that were not I-confluent took on a range of forms. Three validations performed the equivalent of foreign key checking, which, as we have discussed, is unsafe under deletion. Three validations checked database-backed configuration options including the maximum allowed file upload size and default tax rate; while configuration updates are ostensibly rare, the outcome of each validation could be affected under a configuration change. Two validations were especially interesting. Spree's `AvailabilityValidator` checks whether an eCommerce inventory has sufficient stock available to fulfill an order; concurrent order placement might result in negative stock. Discourse's `PostValidator` checks whether a user has been spamming the forum; while not necessarily critical, a spammer could technically foil this validation by attempting to simultaneously author many posts.

In summary, again, a large proportion of validations appear safe. Nevertheless, the few non-I-confluent validations should be cause for concern under concurrent execution.

5. QUANTIFYING FERAL ANOMALIES

While many of the validations we encountered were I-confluent, not all were. In this section, we specifically investigate the effect of concurrent execution on two of the most popular non-I-confluent validations: uniqueness and foreign key validations.

5.1 Uniqueness Constraints and Isolation

To begin, we consider Rails's uniqueness validations: 12.7% of the built-in validation uses we encountered. In this section, we discuss how Rails implements uniqueness and show that this is—at least theoretically—unsafe.

When a model field is declared with a `:validates_uniqueness` annotation, any instance of that model is compared against all other corresponding records in the database to ensure that the field is indeed unique. ActiveRecord accomplishes this by issuing a "`SELECT`" query in SQL and, if no such record is found, Rails updates the instance state in the database (Appendix B.1).

While this user-level uniqueness validation runs within a transaction, the isolation level of the transaction affects its correctness. For correct execution, the `SELECT` query must effectively attain a predicate lock on the validated column for the duration of the transaction. This behavior *is* supported under serializable isolation. However, under Read Committed or Repeatable Read isolation, no such mutual exclusion will be performed, leading to potential inconsistency.[6] Moreover, validation under Snapshot Isolation may similarly result in inconsistencies.[7] Thus, unless the database is configured for serializable isolation, integrity violations may result.

As we have discussed, MySQL and PostgreSQL each support serializable isolation but default to weaker isolation. Moreover, in our investigation, we discovered a bug in PostgreSQL's implementation of Serializable Snapshot Isolation that allowed duplicate records to be created under serializable isolation when running a set of transactions derived from the Rails primary key validator. We have confirmed this anomalous behavior with the core PostgreSQL developers[8] and, as of March 2015, the behavior persists. Thus, any discussion of weak isolation levels aside, PostgreSQL's implementation of serializability is non-serializable and is insufficient to provide correct behavior for Rails' uniqueness validations. So-called "serializable" databases such as Oracle 12c that actually provide Snapshot Isolation will similarly fall prey to duplicate validations.

The Rails documentation warns that uniqueness validations may fail and admit duplicate records [7]. Yet, despite the availability of patches that remedy this behavior by the use of an in-database constraint and/or index, Rails provides this incorrect behavior by default. (One patch was rejected; a developer reports "[t]he reasons for it not being incorporated...are lost in the mists of time but I suspect it's to do with backwards compatibility, cross database compatibility and applications varying on how they want/need to handle these kind of errors." [27]).

In another bug report complaining of duplicates due to concurrent uniqueness validation, a commenter asserts "this is not a bug but documented and inherent behavior of validates_uniqueness_of" [72]. A Rails committer follows up, noting that "the only way to handle [uniqueness] properly is at the database layer with a unique constraint on the column," and subsequently closes the issue. The original bug reporter protests that "the problem extends beyond unique constraints and into validations that are unique to a Rails application that can't [sic?!] be enforced on the DB level"; the Rails committer responds that "with the possible exception of [associations,] all of the other validations are constrained by the attribute values currently in memory, so aren't susceptible to similar flaws." This final statement is correct for many of the built-in validations but is not correct for arbitrary user-defined validations. We discuss the user-defined validation issue further in Section 7.

Understanding validation behavior. Given that entirely feral mechanisms can introduce duplicates, how many duplicates can be introduced? Once a record is written, any later validations will observe it via `SELECT` calls. However, *while* a record is being validated, any number of concurrent validations can unsafely proceed. In practice, the number of concurrent validations is dependent on the Rails environment. In a Rails deployment permitting P concurrent validations (e.g., a single-threaded, multi-process environment with P processes), each value in the domain of the model field/database column can be inserted no more than P times. Thus, validations—at least theoretically—bound the worst-case number of duplicate records for each unique value in the database table.

5.2 Quantifying Uniqueness Anomalies

Given that feral uniqueness validations are acknowledged to be unsafe under non-serializable isolation yet are widely used, we sought to understand exactly how often uniqueness anomalies occur in an experimental deployment. In this section, we demonstrate that uniqueness validations in Rails are indeed unsafe under non-serializable isolation. While they prevent some data integrity errors, we observe—depending on the workload—many duplicate records.

Experimental setup. We developed a Rails 4.1.5 application that performed insertions to a non-indexed string column and compared the incidence of violations both with and without a uniqueness validator (see also Appendix C.1).[9] We deployed this application on two Amazon EC2 `m2.4xlarge` instances, offering 68.4 GB RAM, 8 CPU cores, and 1680GB local storage, running Ubuntu 14.04 LTS. On one instance, we deployed our application, using Nginx 1.6.2 as a web frontend proxied to a set of Unicorn 4.8.3 (Ruby VM pool) workers. Nginx acts as a HTTP frontend and forwards incoming requests to a variably sized pool of Rails VMs (managed by Unicorn, in a multi-process, single-threaded server) that `epoll` on a shared Linux file descriptor. On the other EC2 instance, we deployed PostgreSQL 9.3.5 and configured it to run on the instance local storage. We used a third EC2 instance to direct traffic to the front-end instance and drive load. We plot the average and standard deviation of three runs per experiment.

Stress test. We began our study by issuing a simple stress test that executed a number of concurrent insertion requests against a

[6] Using `SELECT FOR UPDATE` under these weaker models would be safe, but Rails does not implement its predicate-based lookups as such (i.e., it instead opts for a simple `SELECT` statement).

[7] The first reference to the potential integrity violations resulting from this implementation in the Rails code that we are aware of dates to December 2007, in Rails v.2.0.0 [59]. In September 2008, another user added additional discussion within the code comments, noting that "this could even happen if you use transactions with the 'serializable' isolation level" [62]. Without reading too closely, the use of "'serializable"' possibly suggests familiarity with the common, erroneous labeling of Snapshot Isolation as "serializable" (as in Oracle 12c documentation and PostgreSQL documentation prior to the introduction of SSI in version 9.1.1 in September 2011).

[8] "BUG #11732: Non-serializable outcomes under serializable isolation" at `http://www.postgresql.org/message-id/20141021071458.2678.9080@wrigleys.postgresql.org`

[9] In our experimental evaluation, we use the custom applications described below (and in Appendix C) for two reasons. First, these test cases allow us to isolate ActiveRecord behavior to the relevant set of validations as they are deployed by default, independent of any specialized controller logic. Second, this reduces the complexity of automated testing. Many of the applications in our corpus indeed use the same code paths within ActiveRecord, but evaluating these custom applications simplifies programmatic triggering of validation logic.

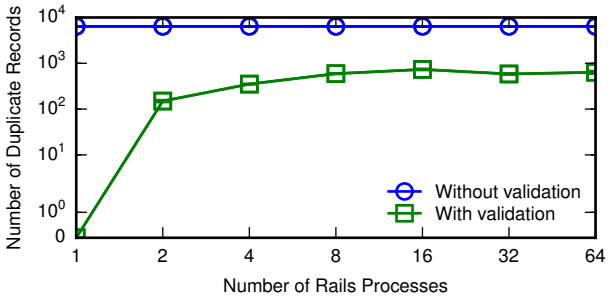

Figure 2: Uniqueness stress test integrity violations.

variable number of Unicorn workers. We repeatedly issued a set of 64 concurrent model creation (SQL insertion) requests, each with the same validated key (e.g., all with field `key` set to value 1) against the Rails application. Across an increasing number of Unicorn workers, we repeated this set of requests 100 times (blocking in-between rounds to ensure that each round is, in fact, a concurrent set of requests), changing the validated key each round (Appendix C.2).

Figure 2 shows the results. With no validation, all concurrent requests succeed, resulting in 6300 duplicate records (100 rounds of 64-1 duplicate keys). With validations enabled, the number of violations depends on the degree of concurrency allowed by Unicorn. With only one process, Unicorn performs the validations serially, creating no duplicates. However, with two processes, Unicorn processes race, resulting in 70 duplicate records spread across 70 keys. With three processes, Unicorn produces 249 duplicate records across all 100 keys. The number of duplicates increases with the number of processes, peaking at 16 workers. With additional workers, duplicate counts decrease slightly, which we attribute to thrashing between workers and within PostgreSQL (recall that each instance has only 8 cores). Nevertheless, using validations, the microbenchmark duplicate count remains below 700—nearly an order-of-magnitude fewer duplicates than without using validations. Therefore, even though these validations are incorrectly implemented, they still result in fewer anomalies. However, when we added in-database unique index on the `key` column[10] and repeated the experiment, we observed no duplicates, as expected.

Actual workloads. The preceding experiment stressed a particularly high-contention workload—in effect, a worst case workload for uniqueness validations. In practice, such a workload is likely rare.[11] Accordingly, we set up another workload meant to capture a less pathological access pattern. We ran another insert-only workload, with key choice distributed among a fixed set of keys. By varying the distribution and number of keys, we were able to both capture more realistic workloads and also control the amount of contention in the workload. As a basis for comparison, we ran four different distributions. First, we considered uniform key access. Second, we used YCSB's Zipfian-distributed accesses from `workloada` [36]. Third and fourth, we used the item distribution access from Facebook's LinkBench workload, which captures MySQL record access

[10] In this case, we added a unique index to the model using Active Record's *database migration*, or manual schema change functionality. Migrations are written separately from the Active Record model declarations. Adding the index was not difficult, but, nevertheless, the index addition logic is separate from the domain model. Without using third-party models, we are unaware of a way to enforce uniqueness within Rails without first declaring an index that is *also* annotated with a special `unique: true` attribute.

[11] In fact, it was in the above workload that we encountered the non-serializable PostgreSQL behavior under serializable isolation. Under serializable isolation, the number of anomalies is reduced compared to the number under Read Committed isolation (as we report here), but we still detected duplicate records.

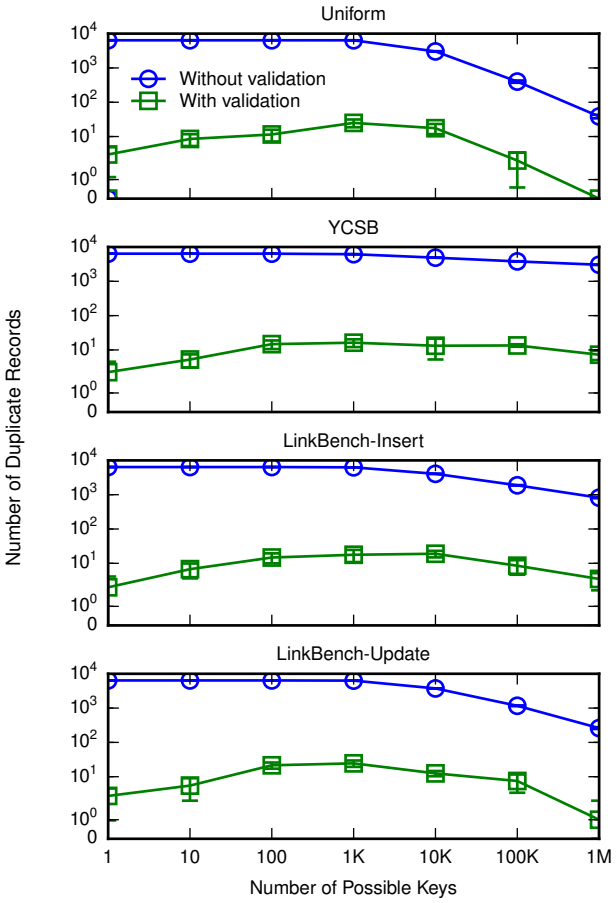

Figure 3: Uniqueness workload integrity violations.

when serving Facebook's social graph [14]. Specifically, we used—separately—the insert and update traffic from this benchmark.

For each trial in this workload, we used 64 concurrent clients independently issuing a set of 100 requests each, with a fixed number of 64 Unicorn workers per process (Appendix C.3).

Figure 3 illustrates the number of duplicate records observed under each of these workloads. As we increase the number of possible keys, there are two opposing effects. With more keys, the probability of any two operations colliding decreases. However, recall that, once a key is written, all subsequent validators can read it. While the uniform workload observes an average of 2.33 duplicate records with only one possible key, it observes an average of 26 duplicate keys with 1000 possible keys. Nevertheless, with 1 million possible keys, we do not observe any duplicate records.

The actual "production" workloads exhibit different trends. In general, YCSB is an extremely high contention workload, with a Zipfian constant of 0.99, resulting in one very hot key. This decreases the beneficial effect of increasing the number of keys in the database. However, LinkBench has less contention and anomalies decrease more rapidly with increased numbers of keys.

5.3 Association Validations and Isolation

Having investigated uniqueness constraints, we turn our attention to association validations. We first, again, discuss how Rails enforces these validations and describe how—at least theoretically—validations might result in integrity errors.

When a model field is declared with an association (e.g., it `:belongs_to` another model) *and* a `:validates_presence` validation, Rails will attempt to ensure that the declared validation is

valid before saving the model. Rails accomplishes this by issuing a "SELECT WHERE" query in SQL to find an associated record (e.g., to ensure the "one" end of a one-to-many relationship exists) and, if a matching association is found, Rails updates the instance state in the database (Appendix B.2). On deletion, any models with associations marked with `:dependent => destroy` (or `:dependent => delete`) will have any associated models destroyed (i.e., removed by instantiating in Rails and calling `destroy` on the model) or deleted (i.e., removed by simply calling the database's `DELETE` method).

This feral association validation runs within a transaction, but, again the exact isolation level of the transaction affects its correctness. For correct execution, the `SELECT` query must also attain a predicate lock on the specific value of the validated column for the duration of the transaction. Similar to the uniqueness validator, concurrent deletions and insertions are unsafe under Read Committed, Repeatable Read, and Snapshot Isolation. Thus, unless the database is configured for serializable isolation, inconsistency may result and the feral validation will fail to prevent data corruption.

Unlike uniqueness validations, there is no discussion of associations and concurrency anomalies in the Rails documentation. Moreover, in Rails 4.1, there is no way to natively declare a foreign key constraint;[12] it must be done via a third-party library such as `foreigner` [57] or `schema_plus` [1]. Only two applications (`canvaslms` and `diaspora`) used `foreigner`, and only one application (`juvia`) used `schema_plus`. One application (`jobsworth`) used a custom schema annotation and constraint generator.

Understanding association behavior. Given that entirely feral mechanisms can introduce broken associations, how many dangling records can be introduced? Once a record is deleted, any later validations will observe it via `SELECT` calls. However, in the worst case, the feral cascading deletion on the one side of a one-to-many relation can stall indefinitely, allowing an unlimited number of concurrent insertions to the many side of the relation. Thus, validations—at least theoretically—only reduce the worst-case number of dangling records that were inserted prior to deletion; any number of concurrent insertions may occur during validation, leading to unbounded numbers of dangling records.

5.4 Quantifying Association Anomalies

Given this potential for errors, we again set out to quantify integrity errors. We demonstrate that weak isolation can indeed lead to data integrity errors in Rails' implementation of associations.

We performed another set of experiments to test association validation behavior under concurrent insertions and deletions. Using the same Unicorn and PostgreSQL deployment as above, we configured another application to test whether or not Rails validations would correctly enforce association-based integrity constraints. We consider an application with two models: Users and Departments. We configure a one-to-many relationship: each user `belongs_to` a department, and each department `has_many` user (Appendix C.4).

As a basic stress test, we initialize the database by creating 100 departments with no users. Subsequently, for each department in the database, we issue a single request to delete the department along with 64 concurrent requests to insert users in that department. To correctly preserve the one-to-many relationship, the database should either reject the deletion operation or perform a cascading deletion of the department and any users (while rejecting any future user creation requests for that department). We can quantify the degree of inconsistency by counting the number of users left in the database who have no corresponding department (Appendix C.5).

[12] Rails 4.2 added support for foreign keys via migration annotation (separate from models; similarly to adding a unique index) in December 2014.

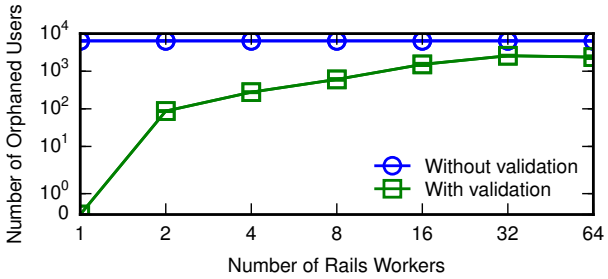

Figure 4: Foreign key stress association anomalies.

With associations declared in Rails, the Rails process performing the deletion will attempt a cascading delete of users upon department deletion. However, this cascade is performed, again, ferally—at the application level. Thus, under non-serializable isolation, any user creation events that are processed while the search for Users to delete is underway will result in Users without departments.

Figure 4 shows the number of "orphaned" Users (i.e., Users without a matching Department) as a function of Rails worker processes. With no constraints declared to Rails or to the database, all User creations succeed, resulting in 6400 dangling Users. With constraints declared in Rails (via a mix of validation and association), the degree of inconsistency depends on the degree of parallelism. Under the worst case, with 64 concurrent processes, the validations are almost worthless in preventing integrity errors. In contrast, when we declare a foreign key constraint within the database[13] and run the workload again, we observe no inconsistency.

The above stress test shows that inconsistency due to feral concurrency control occurs only during times of contention—parallel deletions and insertions. We subsequently varied the degree of contention within the workload. We configured the same application and performed a set of insertions and deletions, but spread across a greater number of keys and at random. A set of 64 processes concurrently each issued 100 User creation and Department deletion requests (at a ratio of 10 to 1) to a set of randomly-selected keys (again at a ratio of 10 Users to each Department). By varying the number of Users and Departments, we were able to control the amount of contention within the workload. Under this workload, inconsistency resulted only when a Department deletion proceeded concurrently with a User creation event and the feral cascading deletion "missed" the User creation (Appendix C.6).

Figure 5 shows the results. As the number of Departments increases, we observe two trends. First, with only one Department, there is again less chance of inconsistency: all operations contend on the same data item, so the total number of inconsistent, orphaned users is limited by the number of potentially racing. However, as the number of Departments increases, the chance of concurrent deletions and insertions drops.

5.5 Takeaways and Discussion

The preceding experiments demonstrate that, indeed, Active Record is unsafe as deployed by default. Validations are susceptible to data corruption due to sensitivity to weak isolation anomalies.

This raises the question: why declare validations at all? As we observe, validations protect against *some* data corruption. First, they correctly guard against non-concurrency-related anomalies such as data entry or input errors. For example, if a user attempts to reserve a username that was previously chosen, a validation would succeed.

[13] In this case, we introduced the constraint via SQL using a direct connection to the database. This change was straightforward but—like the unique index addition—was not reflected in the base Active Record models.

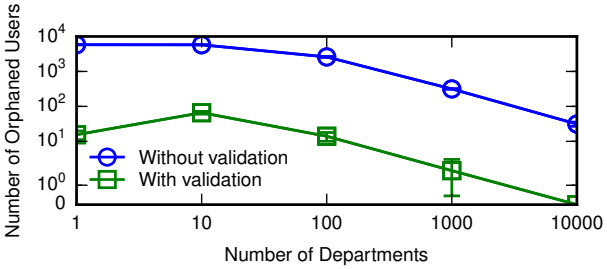

Figure 5: Foreign key workload association anomalies.

The failures we observe here are solely due to concurrent execution. Without concurrent execution, validations are correct. Second, validations *do* reduce the incidence of inconsistency. Empirically, even under worst-case workloads, these validations result in order-of-magnitude reductions in inconsistency. Under less pathological workloads, they may eliminate it. It is possible that, in fact, the degree of concurrency and data contention within Rails-backed applications simply does not lead to these concurrency races—that, in some sense, validations are "good enough" for many applications.

Nevertheless, in both cases, Rails's feral mechanisms are a poor substitute for their respective database counterparts—at least in terms of integrity. We re-examine the Rails community's reluctance to embrace these mechanisms in Section 7.

6. OTHER FRAMEWORKS

While our primary focus in this paper is Rails, we briefly investigated support for uniqueness, foreign key, and custom validations in several other ORM frameworks. We find widespread support for validations and varying susceptibility to integrity errors.

Java Persistence API (JPA; version EE 7) [3] is a standard Java Object persistence interface and supports both uniqueness and primary key constraints in the database via specialized object annotations. Thus, when JPA is used to create a table, it will use the database to enforce these constraints. In 2009, JPA introduced support for UDF validations via a JavaBean interface [23]. Interestingly, both the original (and current) Bean validation specifications specifically address the use of uniqueness validations in their notes:

> "Question: should we add @Unique that would map to @Column(unique=true)? @Unique cannot be tested at the Java level reliably but could generate a database unique constraint generation. @Unique is not part of the [Bean Validation] spec today." [21]

An author of a portion of the code specification notes separately:

> "The reason @Unique is not part of the built-in constraints is the fact that accessing the [database] during a valiation [sic] is opening yourself up for potenital [sic] phantom reads. Think twice before you go for [an application-level] approach." [46]

By default, JPA Validations are run upon model save and run in a transaction at the default isolation level, and therefore, as the developers above hint, are susceptible to the same kinds of integrity violations we study here.

Hibernate (version 4.3.7) [56], a Java ORM based on JPA, does *not* automatically enforce declared foreign key relationships: if a foreign key constraint is declared; a corresponding column is added, but that column is *not* backed by a database foreign key. Instead, for both uniqueness and foreign key constraints, Hibernate relies on JPA schema annotations for correctness. Therefore, without appropriate schema annotations, Hibernate's basic associations may contain dangling references. Hibernate also has an extensive user-level validation framework implementing the JPA Validation Bean specification [47] and is sensitive to weak isolation anomalies, similar to Rails validations.

CakePHP (version 2.5.5) [4], a PHP-based web framework, supports uniqueness, foreign key, and UDF validations. CakePHP does *not* back any of its validation checking with a database transaction and relies on the user to correctly specify any corresponding foreign keys or uniqueness constraints within the database the schema. Thus, while users can declare each of these validations, there is no guarantee that they are actually enforced by the database. Thus, unless users are careful to specify constraints in both their schema and in their validations, validations may lead to integrity violations.

Laravel (version 4.2) [6], another PHP-based web framework, supports the same set of functionality as CakePHP, including application-level uniqueness, foreign key, and UDF validations in the application. Any database-backed constraints must be specified manually in the schema. Per one set of community documentation [43], "database-level validations can efficiently handle some things (such as uniqueness of a column in heavily-used tables) that can be difficult to implement otherwise" but "testing and maintenance is more difficult...[and] your validations would be database- and schema-specific, which makes migrations or switching to another database backend more difficult in the future." In contrast, model-level validations are "the recommended way to ensure that only valid data is saved into your database. They are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain."

Django (version 1.7) [5], a popular Python-based framework, backs declared uniqueness and foreign key constraints with database-level constraints. It also supports custom validations, but these validations are not wrapped in a transaction [15]. Thus, Django also appears problematic, but only for custom validations.

Waterline (version 0.10) [20], the default ORM for Sails.js (a popular MVC framework for Node.js [19]), provides support for in-DB foreign key and uniqueness constraints (when supported by the database) as well as custom validations (that are *not* supported via transactions; e.g., "TO-DO: This should all be wrapped in a transaction. That's coming next but for the meantime just hope we don't get in a nasty state where the operation fails!" [78]).

Summary. In all, we observe common cross-framework support for feral validation/invariants, with inconsistent use of mechanisms for enforcing them, ranging from the use of in-database constraints to transactions to no ostensible use of concurrency control in either application or database.

7. IMPLICATIONS FOR DATABASES

In light of this empirical evidence of the continued mismatch between ORM applications and databases, in this section, we reflect on the core database limitations for application writers today and suggest a set of directions for alleviating them.

7.1 Summary: Shortcomings Today

The use of feral invariants is not well-supported by today's databases. At a high level, today's databases effectively offer two primary options for ORM framework developers and users:

1. **Use ACID transactions.** Serializable transactions are sufficient to correctly enforce arbitrary application invariants, including transaction-backed feral validations. This is core to the transaction concept: isolation is a means towards preserving integrity.

Unfortunately, in practice, for application developers, transactions are problematic. Given serializability's performance and availability overheads [26], developers at scale have largely eschewed the use of serializable transactions (which are anyway not required for correct enforcement of approximately 75% of the invariants we encountered in the Rails corpus). Moreover, many databases offering "ACID" semantics do not provide serializability by default and often, even among industry-standard enterprise offerings, do not offer it as an option at all [16] (to say nothing of implementation difficulties, as in Footnote 8). Instead, developers using these systems today must manually reason about a host of highly technical, often obscure, and poorly understood weak isolation models expressed in terms of low-level read/write anomalies such as Write Skew and Lost Update [8, 13]. We have observed (e.g., Footnote 7) that ORM and expert application developers are familiar with the prevalence of weak isolation, which may also help explain the relative unpopularity of transactions within the web programming community.

2. **Custom, feral enforcement.** Building user-level concurrency control solutions on a per-framework or, worse, per-application basis is an expensive, error-prone, and difficult process that neglects decades of contributions from the database community. While this solution is sufficient to maintain correctness in the approximately 87% (I-confluent) invariants in our corpus, the remainder can—in many modern ORM implementations—lead to data corruption on behalf of applications.

   However, and perhaps most importantly, this feral approach preserves a key tenet of the Rails philosophy: a recurring insistence on expressing domain logic in the application. This also enables the declaration of invariants that are not among the few natively supported by databases today (e.g., uniqueness constraints).

In summary, application writers today lack a solution that guarantees correctness while maintaining high performance *and* programmability. Serializability is too expensive for some applications, is not widely supported, and is not necessary for many application invariants. Feral concurrency control is often less expensive and is trivially portable but is not sufficient for many other application invariants. In neither case does the database respect and assist with application programmer desires for a clean, idiomatic means of expressing correctness criteria in domain logic. We believe there is an opportunity and pressing need to build systems that provide all three criteria: performance, correctness, and programmability.

7.2 Domesticating Feral Mechanisms

Constructively, to properly provide database support and thereby "domesticate" these feral mechanisms, we believe application users and framework authors need a new database interface that will enable them to:

1. *Express correctness criteria in the language of their domain model, with minimal friction, while permitting their automatic enforcement.* Per Section 2, a core factor behind the success of ORMs like Rails appears to be their promulgation of an idiomatic programming style that "seems right" for web programming. Rails' disregard for advanced database functionality is evidence of a continued impedance mismatch between application domain logic and current database primitives: databases today do not understand the semantics of feral validations.

   We believe any solution to domestication must respect ORM application patterns and programming style, including the ability to specify invariants in each framework's native language. Ideally, database systems could enforce applications' existing feral invariants without modification. This is already feasible for a subset of invariants—like uniqueness and foreign key constraints—but not all. An ideal solution to domestication would provide universal support with no additional overhead for application writers.

2. *Only pay the price of coordination when necessary.* Per Section 4, many invariants can be safely executed without coordination, while others cannot. The many that do not need coordination should not be unnecessarily penalized.

   An ideal solution to domestication would enable applications to avoid coordination whenever possible, thus maximizing both performance and operation availability. The database should facilitate this avoidance, thus evading common complaints (especially within the Internet community) about serializable transactions.

3. *Easily deploy to multiple database backends.* ORM frameworks today are deployed across a range of database implementations, and, when deciding which database features to exercise, framework authors often choose the least common denominator for compatibility purposes.

   An ideal solution to domestication would preserve this compatibility, possibly by providing a "bolt on" compatibility layer between ORM systems and databases lacking advanced functionality (effectively, a "blessed" set of mechanisms beneath the application/ORM that correctly enforce feral mechanisms).

Fulfilling these design requirements would enable high performance, correct execution, and programmability. However, doing so represents a considerable challenge.

Promise in the literature. The actual vehicle for implementing this interface is an open question, but the literature lends several clues. On the one hand, we do not believe the answer lies in exposing additional read/write isolation or consistency guarantees like Read Committed; these fail our requirement for an abstraction operating the level of domain logic and, as we have noted, are challenging for developers (and researchers) to reason about. On the other hand, more recent proposals for invariant-based concurrency control [17, 65] and a litany of work from prior decades on rule-based [82] and, broadly, semantics-based concurrency control [79] appear immediately applicable and worth (re-)considering. Recent advances in program analysis for extracting invariants [73] and subroutines from imperative code [33] may allow us to programatically suggest new invariants, perform correspondence checking for existing applications, and apply a range of automated optimizations to legacy code [34, 76]. Finally, clean-slate language design and program analysis obviate the need for explicit invariant declaration (thus alleviating concerns of specification completeness) [11, 12, 84]; while adoption within the ORM community is a challenge, we view this exploration as worthwhile.

Summary. In all, the wide gap between research and current practice is both a pressing concern and an exciting opportunity to revisit many decades of research on alternatives to serializability with an eye towards current operating conditions, application demands, and programmer practices. Our proposal here is demanding, but so are the framework and application writers our databases serve. Given the correct primitives, database systems may yet have a role to play in ensuring application integrity.

8. RELATED WORK

There is a large body of related work that we consider in four categories: object relational mapping systems, the study of weak isolation and application requirements, the quantification of isolation behavior, and empirical open source software analysis.

ORMs. Database systems and application programming frameworks have a long history [24, 30, 63]. The "impedance mismatch" between object-oriented programming and the relational model is a perennial problem in data management systems. Ruby on Rails is no exception, and the concurrency control issues we study here are endemic to this mismatch—namely, the disuse of common concurrency control mechanisms like database-backed constraints. Bridging this gap remains an active area of research [66].

The latest wave of web programming frameworks has inspired diverse research spanning databases, verification, and security. StatusQuo uses program analysis and synthesis to transform imperative ORM code into SQL, leveraging the efficiency of database-backed web applications written in the Spring framework [33]. Rails has been the subject of study in the verification of cross-site scripting attacks [32], errors in data modeling of associations [68], and arbitrary, user-specified (non-validation) invariants [25]. Rails-style ORM validations have been used to improve systems security via client-side execution [58, 76]. Our focus here is on the concurrency control requirements and usages of applications written in Rails.

Applications and weak isolation. The issues we examine here are fundamental to the use of weak isolation in data management systems. Non-serializable isolation dates to the mid-1970s [52] and has a colorful history [8]; today, by volume, many database management systems are non-serializable by default [16]. The isolation anomalies surfaced by the stores we study here are directly responsible for violating the integrity of the validations we consider.

However, serializable isolation is not strictly necessary for maintaining application integrity. Semantic-based concurrency control criteria has almost as long a lineage as serializability [44, 53] and suggests that, with additional, non-syntactic knowledge about applications (e.g., integrity constraints) [61], correctness is achievable without serializability. This use of invariants has enjoyed recent popularity in work by Li et al. [65], Roy et al. [73], and Bailis et al. [17]. We use the concept of invariant confluence from [17] to determine whether Rails's built-in validators and applications written in Rails are indeed safe under any coordination-free execution. Our methodology is closest in spirit to [17], but, here, we examine real applications instead of standardized benchmarks.

Quantifying anomalies. A range of research similarly quantifies the effect of non-serializable isolation in a variety of ways.

Perhaps closest to our work is a study by Fekete et al., which quantitatively analyzed data inconsistencies arising from non-serializable schedules [45]. This study used a hand-crafted benchmark for analysis but is nevertheless one of the only studies of actual application inconsistencies. Here, we focus on open source applications from the Rails community.

A larger body of work examines isolation anomalies at the read-write interface (that is, measures deviations from properties such as serializability or linearizability but *not* the end effect of these deviations on actual application behavior). Wada et al. evaluated the staleness of Amazon's SimpleDB using end-user request tracing [81], while Bermbach and Tai evaluated Amazon S3 [22], each quantifying various forms of non-serializable behavior. Golab et al. provide algorithms for verifying the linearizability of and sequential consistency arbitrary data stores [51] and Zellag and Kemme provide algorithms for verifying their serializability [85] and other cycle-based isolation anomalies [86]. Probabilistically Bounded Staleness provides time- and version-based staleness predictions for eventually consistent data stores [18]. Our focus here is on anomalies as observed by application logic rather than read-write anomalies observed under weak isolation.

Empirical software analysis. Empirical software analysis of open source software is a topic of active interest in the software engineering research community [77]. In the parlance of that community, in this work, we perform a mixed-methods analysis, combining quantitative survey techniques with a confirmatory case study of Rails's susceptibility to validation errors [42]. In our survey, we attempt to minimize sampling bias towards validation-heavy projects by focusing our attention on popular projects, as measured by GitHub stars. Our use of quantitative data followed by supporting qualitative data from documentation and issue tracking—as well as the chronology of methodologies we employed to attain the results presented here—can be considered an instance of the sequential exploration strategy [37]. We specifically use these techniques in service of better understanding use of database concurrency control.

9. CONCLUSIONS

In this work, we examined the use of concurrency control mechanisms in a set of 67 open source Ruby on Rails applications and, to a less thorough extent, concurrency control support in a range of other web-oriented ORM frameworks. We found that, in contrast with traditional transaction processing, these applications overwhelmingly prefer to leverage application-level *feral* support for data integrity, typically in the form of declarative (sometimes user-defined) validation and association logic. Despite the popularity of these invariants, we find limited use of in-database support to correctly implement them, leading to a range of quantifiable inconsistencies for Rails' built-in uniqueness and association validations. While many validations are invariant confluent and therefore correct under concurrent execution given standard RDBMS weak isolation and concurrent update semantics, we see considerable opportunity to better support these users and their feral invariants in the future.

Coda: A Call for Empiricism

This work is a first step towards better understanding how users in the wild actually interact with the database systems that this community builds. Given the ascendancy of open source, there is unprecedented opportunity to empirically and quantitatively study how our systems are and are not serving the needs of application programmers. Lightweight program analysis has never been easier, and the corpus of readily-accessible code—especially in an academic context—has never been larger.

These open source applications are undoubtedly dwarfed by many other commercial and enterprise-grade codebases in terms of size, quality, and complexity. However, compared to alternatives such as TPC-C, which today is almost 23 years old and is still the preferred standard for transaction processing evaluation, open source corpuses are far better proxies for modern applications. Recent efforts like the OLTPBenchmark suite [39] are promising but are nevertheless (and perhaps necessarily) not a substitute for real applications. The opportunity to perform both quantitative surveys across a large set of applications as well as longitudinal studies over the history of each application repository (and the behavior of a given programmer over time and across repositories) is particularly compelling. While these studies are inherently imprecise (due to limitations of the corpuses), the resulting quantitative trends are invaluable.

In summary, in this era of "Big Data" analytics, we see great promise in turning these analyses inwards, towards an empirical understanding of the usage of data management systems today, in service of better problem selection and a more quantitatively informed community dialogue.

Acknowledgments. The authors would like to thank Peter Alvaro, Michael R. Bernstein, Colin Jones, Xavier Shay, and the SIGMOD reviewers for their insightful commentary and feedback on this

work. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, DARPA XData Award FA8750-12-2-0331, the NSF Graduate Research Fellowship (grant DGE-1106400), and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata, VMware, and Yahoo!.

10. REFERENCES

[1] SchemaPlus. https://github.com/SchemaPlus/schema_plus.

[2] How a quiet developer built Goodreads.com into book community of 2.6+ million members – with Otis Chandler, November 2009. http://mixergy.com/interviews/goodreads-otis-chandler/.

[3] Java EE 7 API: Package javax.persistence, 2013. http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html.

[4] CakePHP, 2014. http://cakephp.org/ and http://book.cakephp.org/2.0/en/index.html.

[5] Django: The Web framework for perfectionists with deadlines, 2014. https://www.djangoproject.com/ and https://github.com/django/django.

[6] Laravel: The PHP Framework for Web Artisans, 2014. http://laravel.com/ and https://github.com/laravel/laravel.

[7] RailsGuide: Active Record Validations, 2014. http://guides.rubyonrails.org/active_record_validations.html.

[8] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[9] R. Allen. Airbnb Engineering Blog: "Upgrading Airbnb from Rails 2.3 to Rails 3.0", October 2012. http://nerds.airbnb.com/upgrading-airbnb-from-rails-23-to-rails-30/.

[10] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services*. Springer, 2004.

[11] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *ICDE 2014*.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, and W. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011*.

[13] P. Alvaro et al. Consistency without borders. In *SoCC 2013*.

[14] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the Facebook social graph. In *SIGMOD 2013*.

[15] J. Aylett. django-database-constraints, 2013. https://github.com/jaylett/django-database-constraints.

[16] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB 2014*.

[17] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Coordination avoidance in database systems. In *VLDB 2015*.

[18] P. Bailis, S. Venkataraman, M. J. Franklin, et al. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB 2012*.

[19] Balderdash. Sails.js: Realtime MVC Framework for Node.js, 2014. https://github.com/balderdashy/sails.

[20] Balderdash. Waterline: An adapter-based ORM for Node.js with support for mysql, mongo, postgres, redis, [sic] and more, 2014. https://github.com/balderdashy/waterline.

[21] Bean Validation Expert Group. Jsr-000303 bean validation 1.0 final release specification, 2009. http://download.oracle.com/otndocs/jcp/bean_validation-1.0-fr-oth-JSpec/.

[22] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *MW4SOC*, 2011.

[23] E. Bernard. Java Specification Request 349: Bean Validation 1.1, 2013. https://jcp.org/en/jsr/detail?id=349.

[24] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.

[25] I. Bocić and T. Bultan. Inductive verification of data model invariants for web applications. In *ICSE*, ICSE 2014, pages 620–631, New York, NY, USA, 2014. ACM.

[26] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[27] J. Brough. #645: Alternative to validates_uniqueness_of using db constraints, 2011. rails/rails at https://github.com/rails/rails/issues/645.

[28] P. Calcado. Building products at SoundCloud – Part I: Dealing with the monolith, June 2014. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[29] M. J. Carey and D. J. DeWitt. Of objects and databases: A decade of turmoil. In *VLDB*, 1996.

[30] M. J. Carey et al. Shoring up persistent applications. In *SIGMOD 1994*.

[31] A. Carter. Hulu Tech Blog: "At a glance: Hulu hits Rails Conf 2012", May 2012. http://tech.hulu.com/blog/2012/05/14/347/.

[32] A. Chaudhuri and J. S. Foster. Symbolic security analysis of Ruby-on-Rails web applications. In *CCS*, 2010.

[33] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR 2013*.

[34] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. In *VLDB 2012*.

[35] B. Cook. Scaling Twitter, SDForum Silicon Valley Ruby Conference, 2007. http://www.slideshare.net/Blaine/scaling-twitter.

[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SoCC 2010*.

[37] J. W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage, 2013.

[38] K. Crum. #3238: Activerecord::staleobjecterror in checkout, 2013. spree/spree at https://github.com/spree/spree/issues/3238.

[39] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *VLDB 2014*.

[40] J. Duff. How Shopify scales Rails, Big Ruby 2013, April 2013. http://www.slideshare.net/jduff/how-shopify-scales-rails-20443485.

[41] E. Dumbill. O'Reilly: "Ruby on Rails: An interview with David Heinemeier Hansson", August 2005. http://www.oreillynet.com/pub/a/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html.

[42] S. Easterbrook et al. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[43] M. Ehsan. Input validation with Laravel, 2014. http://laravelbook.com/laravel-input-validation/.

[44] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[45] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. In *VLDB 2009*.

[46] H. Ferentschik. Accessing the Hibernate Session within a ConstraintValidator, May 2010. https://developer.jboss.org/wiki/AccessingtheHibernateSessionwithinaConstraintValidator.

[47] H. Ferentschik and G. Morling. Hibernate validator JSR 349 reference implementation 5.1.3.final, 2014. https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/.

[48] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[50] D. Geer. Will software developers ride Ruby on Rails to success? *Computer*, 39(2):18–20, 2006.

[51] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *PODC 2011*.

[52] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[53] P. W. Grefen and P. M. Apers. Integrity control in relational database systems–an overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[54] D. H. Hansson. active_record/transactions.rb, 2004. rails/rails githash db045db at https://github.com/rails/rails/blob/db045dbb.

[55] D. H. Hansson. Choose a single layer of cleverness, September 2005. http://david.heinemeierhansson.com/arc/2005_09.html.

[56] Hibernate Team and JBoss Visual Design Team. Hibernate reference documentation 4.3.7.final, 2014. http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/.

[57] M. Higgins. Foreigner. https://github.com/matthuhiggins/foreigner.

[58] T. Hinrichs et al. Caveat: Facilitating interactive and secure client-side validators for ruby on rails applications. In *SECURWARE 2013*.

[59] M. Koziarski. Warn users about the race condition in validates_uniqueness_of. [koz], 2007. rails/rails githash c01c28c at https://github.com/rails/rails/commit/c01c28c.

[60] G. E. Krasner, S. T. Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[61] H.-T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, 1979.

[62] H. Lai. Document concurrency issues in validates_uniqueness_of., 2008. rails/rails githash adacd94 at https://github.com/rails/rails/commit/adacd94.

[63] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[64] J. Leighton. Support for specifying transaction isolation level, 2012. rails/rails githash 392eeec at https://github.com/rails/rails/commit/392eeec.

[65] C. Li, J. Leitao, A. Clement, N. Preguiça, R. Rodrigues, et al. Automating the choice of consistency levels in replicated systems. In *USENIX ATC 2014*.

[66] A. Malpani et al. Reverse engineering models from databases to bootstrap application development. In *ICDE*, 2010.

[67] S. McCullough. Groupon Engineering Blog: "Geekon: I-Tier", October 2013. `https://engineering.groupon.com/2013/node-js/geekon-i-tier/`.

[68] J. Nijjar and T. Bultan. Bounded verification of ruby on rails data models. In *ACM ISSTA*, 2011.

[69] C. Nutter. Q/a: What thread-safe Rails means, August 2008. `http://blog.headius.com/2008/08/qa-what-thread-safe-rails-means.html`.

[70] T. Preston-Werner. How we made GitHub fast, October 2009. `https://github.com/blog/530-how-we-made-github-fast`.

[71] J. Rizzo. Twitch: The official blog "Technically Speaking – Group Chat and General Chat Engineering", April 2014. `http://blog.twitch.tv/2014/04/technically-speaking-group-chat-and-general-chat-engineering/`.

[72] D. Roberts. #13234: Rails concurrency bug on save, 2013. rails/rails at `https://github.com/rails/rails/issues/13234`.

[73] S. Roy, L. Kot, et al. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[74] S. Ruby, D. Thomas, D. H. Hansson, et al. *Agile web development with Rails 4*. The Pragmatic Bookshelf, Dallas, Texas, 2013.

[75] N. Singh. update_attributes and update_attributes! are now wrapped in a transaction, 2010. rails/rails githash f4fbc2c at `https://github.com/rails/rails/commit/f4fbc2c`.

[76] N. Skrupsky et al. Waves: Automatic synthesis of client-side validation code for web applications. In *IEEE CyberSecurity 2012*.

[77] K.-J. Stol et al. The use of empirical methods in open source software research: Facts, trends and future directions. In *FLOSS*, 2009.

[78] C. Stoltman. initial stab at creating has_many relationships, 2013. balderdashy/waterline githash b05fb1c at `https://github.com/balderdashy/waterline/commit/b05fb1c`. As of November 2014, this code has been moved but is still non-transactional and the comment remains unchanged.

[79] M. Tamer Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[80] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications and Strategies*, 65(1):17–37, 2007.

[81] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR 2011*.

[82] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[83] A. Williams. Techcrunch: Zendesk launches a help center that combines self-service with design themes reminiscent of Tumblr, August 2013. `http://techcrunch.com/2013/08/21/zendesk-launches-a-help-center-that-combines-self-service-with-design-themes-reminiscent-of-tumblr/`.

[84] F. Yang et al. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, 2006.

[85] K. Zellag and B. Kemme. How consistent is your cloud application? In *SoCC 2012*.

[86] K. Zellag and B. Kemme. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *ICDE 2011*.

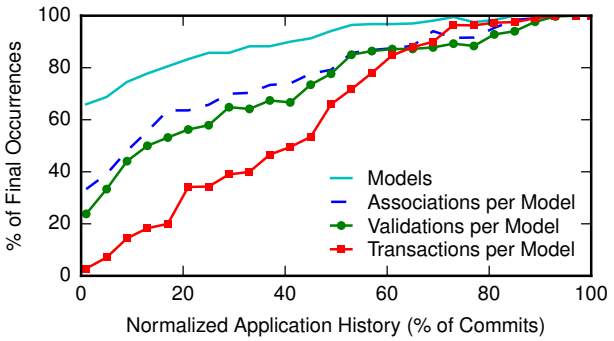

Figure 6: Use of mechanisms over each project's history. We plot the median value of each metric across projects and, for each mechanism, omit projects that do not contain any uses of the mechanism (e.g., if a project lacks transactions, the project is omitted from the median calculation for transactions).

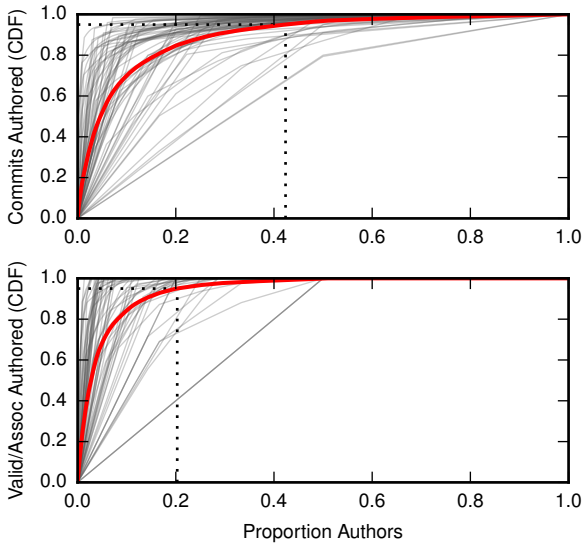

Figure 7: CDFs of authorship of invariants (validations plus associations) and commits. Bolded line shows the average CDF across projects, while faint lines show CDFs for individual projects. The dotted line shows the 95th percentile CDF value.

APPENDIX

A. ANALYSIS METHODOLOGY

To determine the occurrences and number of models, transactions, locks, validations, and associations in Rails, we wrote a set of analysis scripts that performed a very rudimentary syntactic static analysis . We do not consider the analysis techniques here a contribution; rather, our interest is in the output of the analysis. The syntactic approach proved portable between the many versions of Rails against which each application is linked; otherwise, porting between non-backwards-compatible Rails versions was difficult and, in fact, unsupported by several of the Rails code analysis tools we considered using as alternatives. The choice to use syntax as a means of distinguishing code constructs led to some ambiguity. To compensate, we introduced custom logic to handle esoteric syntaxes that arose in particular projects (e.g., some projects extend `ActiveRecord::Base` with a separate, project-specific base class, while some validation usages vary between constructs like `:validates_presence` and `:validates_presence_of`).

To determine code authorship, we used the output of git `log` and `blame` and did not attempt any sophisticated entity resolution.

B. DETAILED VALIDATION BEHAVIOR

B.1 Uniqueness Validation

When a controller attempts to `save` an ActiveRecord model instance i of type M, if M has a declared `:validates_uniqueness` annotation on attribute a, the following actions will be performed:

1. Assuming that instances of M are stored in database table T_M (with attribute a stored in column C_a), Active Record will perform the equivalent of

   `SELECT 1 FROM` T_M `where` C_a `=` $i.a$ `LIMIT ONE;`

   (`SELECT COUNT(*)` would be sufficient here as well, but this is not how the query is actually implemented).

2. If this result set is empty, the validation succeeds.

3. If this result set is not empty, the validation fails. If the validation was called during `save`, it returns `false`. If the validation was called during `save!`, it raises an `ActiveRecord::RecordInvalid` exception.

This is a classic example of the phantom problem. Changing this `SELECT` call to `SELECT FOR UPDATE` would be sufficient. However, Rails is not implemented this way.

Name	Description	Authors	LoC Ruby	Commits	M	T	PL	OL	V	A	Stars	Githash	Last commit
Canvas LMS	Education	132	309,580	12,853	161	46	12	1	354	837	1,251	3fb8e69	10/16/14
OpenCongress	Congress data	15	30,867	1,884	106	1	0	0	48	357	124	850b602	02/11/13
Fedena	Education management	4	49,297	1,471	104	5	0	0	153	317	262	40cafe3	01/23/13
Discourse	Community discussion	440	72,225	11,480	77	41	0	0	83	266	12,233	1cf4a0d	10/20/14
Spree	eCommerce	677	47,268	14,096	72	6	0	0	92	252	5,582	aa34b3a	10/16/14
Sharetribe	Content management	35	31,164	7,140	68	0	0	0	112	202	127	8e0d382	10/21/14
ROR Ecommerce	eCommerce	19	16,808	1,604	63	2	3	0	219	207	857	c60a675	10/09/14
Diaspora	Social network	388	31,726	14,640	63	2	0	0	66	128	9,571	1913397	10/03/14
Redmine	Project management	10	81,536	11,042	62	11	0	1	131	157	2,264	e23d4d9	10/19/14
ChiliProject	Project management	53	66,683	5,532	61	7	0	1	118	130	623	984c9ff	08/13/13
Spot.us	Community reporting	46	94,705	9,280	58	0	0	0	96	165	343	61b65b6	12/02/13
Jobsworth	Project management	46	24,731	7,890	55	10	0	0	86	225	478	3a1f8e1	09/12/14
OpenProject	Project management	63	84,374	11,185	49	8	1	3	136	227	371	c1e66af	11/21/13
Danbooru	Image board	25	27,857	3,738	47	9	0	0	71	114	238	c082ed1	10/17/14
Salor Retail	Point of Sale	26	18,404	2,259	44	0	0	0	81	309	24	00e1839	10/07/14
Zena	Content management	7	56,430	2,514	44	1	0	0	12	43	172	79576ac	08/18/14
Skyline CMS	Content management	7	10,404	894	40	5	0	0	28	89	127	64b0932	12/09/13
Opal	Project management	6	10,707	474	38	3	0	0	42	96	45	11edf34	01/09/13
OneBody	Church portal	33	20,398	3,973	36	3	0	0	97	140	1,041	2dfbd4d	10/19/14
CommunityEngine	Social networking	67	13,967	1,613	35	3	0	0	92	101	1,073	a4d3ea2	10/16/14
Publify	Blogging	93	16,763	5,067	35	7	0	0	33	50	1,274	4acf86e	10/20/14
Comas	Conference management	5	5,879	435	33	6	0	0	80	45	21	81c25a4	09/09/14
BrowserCMS	Content management	56	21,259	2,503	32	4	0	0	47	77	1,183	d654557	09/30/14
RailsCollab	Project managment	25	8,849	865	29	6	0	0	40	122	262	9f6c8c1	02/16/12
OpenGovernment	Government data	15	9,383	2,231	28	4	0	0	22	141	160	fa80204	11/21/13
Tracks	Personal productivity	89	17,419	3,121	27	2	0	0	24	43	639	eb2650c	10/02/14
GitLab	Code management	671	39,094	12,266	24	15	0	0	131	114	14,129	72abe9f	10/20/14
Brevidy	Video sharing	2	7,608	6	24	1	0	0	74	56	167	d0ddb1a	01/18/14
Insoshi	Social network	16	121,552	1,321	24	1	0	0	41	63	1,583	9976cfe	02/24/10
Alchemy	Content management	34	19,329	4,222	23	2	0	0	37	40	240	91d9d08	10/20/14
Teambox	Project management	48	32,844	3,155	22	2	0	0	56	116	1,864	62a8b02	09/20/11
Fat Free CRM	Customer relationship	99	21,284	4,144	21	3	0	0	39	92	2,384	3dd2c62	10/17/14
linuxfr.org	FLOSS community	29	8,123	2,271	20	1	0	0	50	50	86	5d4d6df	10/14/14
Squash	Bug reporting	28	15,776	231	19	6	0	0	87	62	879	c217ac1	09/15/14
Shoppe	eCommerce	14	3,172	349	19	1	0	0	58	34	208	19e60c8	10/18/14
nimbleShop	eCommerce	12	8,041	1,805	19	0	0	0	47	34	47	4254806	02/18/13
Piggybak	eCommerce	16	2,235	383	17	1	0	0	51	35	166	2bed094	09/10/14
wallgig	Wallpaper sharing	6	5,543	350	17	1	0	0	42	45	18	4424d44	03/23/14
Rucksack	Collaboration	7	5,346	445	17	3	0	0	18	79	169	59703d3	10/05/13
Calagator	Online calendar	48	9,061	1,766	16	0	0	0	8	11	196	6e5df08	10/19/14
Amahi Platform	Home media sharing	15	6,244	577	15	2	0	0	38	22	65	5101c8b	08/20/14
Sprint	Project management	5	3,056	71	14	0	0	0	50	45	247	584d887	09/17/14
Citizenry	Community directory	17	8,197	512	13	0	0	0	12	45	138	e314fe4	04/01/14
LovdByLess	Social network	17	30,718	150	12	0	0	0	27	41	568	26e79a7	10/09/09
lobste.rs	Link sharing	24	4,963	624	12	8	0	0	20	40	646	b0b9654	10/18/14
BucketWise	Personal finance	10	4,644	258	12	2	0	0	11	46	484	5c73f2b	06/10/12
Sugar	Forum	13	7,703	1,316	11	1	0	0	20	53	89	49ca79f	10/21/14
Comf. Mexican Sofa	Content management	106	8,881	1,746	10	0	0	0	35	26	1,523	fecef0c	10/09/14
Radiant	Content management	100	15,923	2,385	9	3	0	1	26	12	1,554	0c9ef9b	10/01/14
Forem	Forum	100	4,676	1,383	9	0	0	0	8	29	1,302	519f2de	08/14/14
Saasy	eCommerce	2	163,170	21	8	4	0	0	19	9	520	4fe610f	08/03/09
Refinery CMS	Content management	438	10,847	9,107	8	0	0	0	16	8	2,979	f4e24ef	10/20/14
BostonRB	Ruby community	40	2,135	889	7	0	0	0	18	12	199	05fc100	10/21/14
Inkwell	Social networking	6	6,764	156	7	0	0	0	4	51	327	d1938d3	07/15/14
Boxroom	File sharing	9	1,956	368	6	0	0	0	18	12	218	1e74e06	10/18/14
Copycopter	Copy writing	9	2,347	46	6	1	0	0	7	14	652	d3607c4	06/28/12
Enki	Blogging	29	4,678	562	6	1	0	0	5	7	835	b793d48	12/01/13
Fulcrum	Project planning	46	3,190	637	5	0	0	0	13	15	1,335	8397de2	08/20/14
GitLab CI	Continuous integration	80	3,700	870	5	2	0	0	11	13	1,188	7d51134	10/17/14
Kandan	Persistent chat	56	1,694	808	5	0	0	0	6	8	2,249	15a8aab	10/06/14
Juvia	Commenting	8	2,302	202	4	3	0	0	11	8	937	43a1c48	05/09/14
Go vs Go	Go board game	2	2,378	302	4	0	0	0	11	9	145	c8d739d	02/21/13
Adopt-a-Hydrant	Civics	14	14,165	1,242	3	0	0	0	11	8	182	5b7ea0e	10/21/14
Selfstarter	Crowdfunding	23	577	127	3	0	0	0	1	4	2,688	740075f	05/16/14
Heaven	Code deployment	19	2,090	387	2	0	0	0	2	2	163	2d4162e	10/21/14
Carter	eCommerce	3	1,093	70	2	1	0	0	0	12	22	60ad49d	07/22/14
Obtvse	Blogging	27	455	393	1	0	0	0	3	0	1,516	1542856	03/21/13
Average:		**69.10**	**26,809.51**	**2,950.85**	**29.07**	**3.84**	**0.24**	**0.10**	**52.31**	**92.87**	**1,272.42**		**02/06/14**

Table 2: Corpus of applications used in analysis (M: Models, T: Transactions, PL: Pessimistic Locking, OL: Optimistic Locking, V: Validations, A: Associations). Stars record number of GitHub Stars as of October 2014.

B.2 Association Validation

When a controller attempts to save an ActiveRecord model instance i of type M, if M has a declared :belongs_to annotation on attribute a pointing to attribute b of model N *and* M has a declared :validates_presence annotation on attribute a, the following actions will be performed:

1. Assuming that instances of N are stored in database table T_N (with attribute b stored in column C_b), Active Record will perform the equivalent of

   SELECT 1 FROM T_N where C_b = $i.a$ LIMIT ONE;

2. If this result set is not empty, the validation succeeds.
3. If this result set is empty, the validation fails. If the validation was called during save, it returns false. If the validation was called during save!, it raises an ActiveRecord::RecordInvalid exception.

C. EXPERIMENTAL DESCRIPTION

We describe our applications from Section 5 in greater detail.

C.1 Uniqueness Validation Schema

We declare two models, each containing two attributes: key, a string, and value, also a string. The generated schema for each of the models, which we call SimpleKeyValue and ValidatedKeyValue, is the same. The schema for SimpleKeyValue is as follows:

```
create_table "validated_key_values", force: true do |t|
  t.string "key"
  t.string "value"
  t.datetime "created_at"
  t.datetime "updated_at"
end
```

For the non-uniqueness-validated model, we simply require that the key and value fields are not null via a presence: true annotation. For the ferally validated model, we add an additional uniqueness: true validation to the key field in the Active Record model. The remainder of the application consists of a simple View and Controller logic to allow us to POST, GET, and DELETE each kind of model instance programatically via HTTP.

C.2 Uniqueness Stress Test

For the uniqueness stress test (Figure 2), we repeatedly attempt to create duplicate records. We issue a set of 64 concurrent requests to create instances with the key field set to an increasing sequence number (k, below) and repeat 100 times. At the end of the run, we count the number of duplicate records in the table:

for model $m \in$ {SimpleKeyValue, ValidatedKeyValue} **do**
  for $k \leftarrow 1$ to 100 **do**
    parfor 1 to 64 **do**
      via HTTP: create new m with key=k
  dups ←execute(SELECT key, COUNT(key)-1 FROM T_M
    GROUP BY key HAVING COUNT(key) > 1;)

Under correct validation, for each choice of k (i.e., for each key k), all but one of the model creation requests should fail.

C.3 Uniqueness Workload Test

For the uniqueness workload test (Figure 3), a set of 64 workers sequentially issues a set of 100 operations each. Each operation attempts to create a new model instance with the key field set to a random item generated according to the distributions described in Section 5:

for model $m \in$ {SimpleKeyValue, ValidatedKeyValue} **do**
  parfor 1 to 64 **do**
    for 1 to 100 **do**
      $k \leftarrow$ *pick new key according to distribution*
      via HTTP: create new m with key=k
  dups ←execute(SELECT key, COUNT(key)-1 FROM T_M
    GROUP BY key HAVING COUNT(key) > 1;)

C.4 Association Validation Schema

We declare two sets of models, each containing two models each: a User model and a Departments model. Each User has a(n implicit) id (as generated by Rails ActiveRecord) and an integer corresponding department_i. Each Department has an id. Both models have a timestamp of the last updated and creation time, as is auto-generated by Rails. Aside from the table names, both schemas are equivalent. Below is the schema for the non-validated users and departments:

```
create_table "simple_users", force: true do |t|
    t.integer "simple_department_id"
    t.datetime "created_at"
    t.datetime "updated_at"
end

create_table "simple_departments", force: true do |t|
    t.datetime "created_at"
    t.datetime "updated_at"
end
```

The two pairs of models vary in their validations. One pair of models has no validations or associations. The other pair of models contain validations, including rules for cascading deletions. Specifically, we place an association has_many :users, :dependent => :destroy on the department, and, on the user, an association belongs_to :department and validation validates :department, :presence => true (note that we only delete from Departments in our workload, below). Thus, on deletion of a model of type ValidatedDepartment, ActiveRecord will attempt to call destroy on each matching ValidatedUser.

C.5 Association Stress Test

For the association stress test (Figure 4), we repeatedly attempt to create orphan users. We issue a set of 64 concurrent requests to create Users belonging to a particular department, while simultaneously deleting that department and repeat 100 times. At the end of the run, we count the number of users with a department that does not exist:

for model $m \in$ {Simple, Validated} **do**
  for $i \leftarrow 1$ to 100 **do**
    via HTTP: create mDepartment with id=i
  for $i \leftarrow 1$ to 100 **do**
    parfor $w \in 1$ to 65 **do**
      if $w = 1$ **then**
        via HTTP: delete mDepartment with id=i
      else
        via HTTP: create new mUser department_id=i
  orphaned ←execute("SELECT m_department_id,
    COUNT(*) FROM m_users AS U
    LEFT OUTER JOIN m_departments AS D
    ON U.m_department_id = D.id
    WHERE D.id IS NULL
    GROUP BY m_department_id
    HAVING COUNT(*) > 0;")

C.6 Association Workload Test

For the association workload test (Figure 5), we begin by creating a variable number of departments (Figure 5 x-axis; D). We next have 64 concurrent clients simultaneously attempt to create users belonging to a random department and delete random departments (in a 10:1 ratio of creations to deletions, for 100 operations each). We end by counting the number of orphaned users, as above.

for model $m \in$ {Simple, Validated} **do**
  for $d \leftarrow 1$ to D **do**
    via HTTP: create mDepartment with id=i
  parfor $w \in 1$ to 64 **do**
    $d \leftarrow uniformRandomInt([1, D])$
    if $uniformRandomDouble([0,1]) < \frac{1}{11}$ **then**
      via HTTP: delete mDepartment with id=d
    else
      via HTTP: create new mUser department_id=d
  orphaned ← as above, in stress test