Feral Concurrency

Replacing database native primitives with feral mechanisms may improve maintainability of the application, but does it really work? Are the feral invariants correctly enforced in Rails? Do they work in practice? This paper performs theoretical analysis and emperical studies to answer these questions.

Background

•••

MVC frameworks
Ruby on Rails
ORM-based programming
ActiveRecord

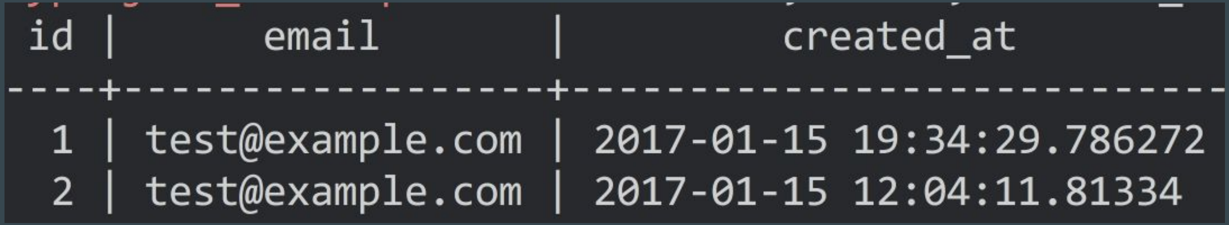
get the database out of the way and let the application do the work.

What happens when you create a user in Rails?

Four queries

- `BEGIN`
- `SELECT * FROM users WHERE email=’foo@bar.com’`
- `INSERT INTO users …`
- `COMMIT`

This is slow
This is also incorrect!



```
 id |       email       |         created_at
----+-------------------+----------------------------
  1 | test@example.com  | 2017-01-15 19:34:29.786272
  2 | test@example.com  | 2017-01-15 12:04:11.81334
```

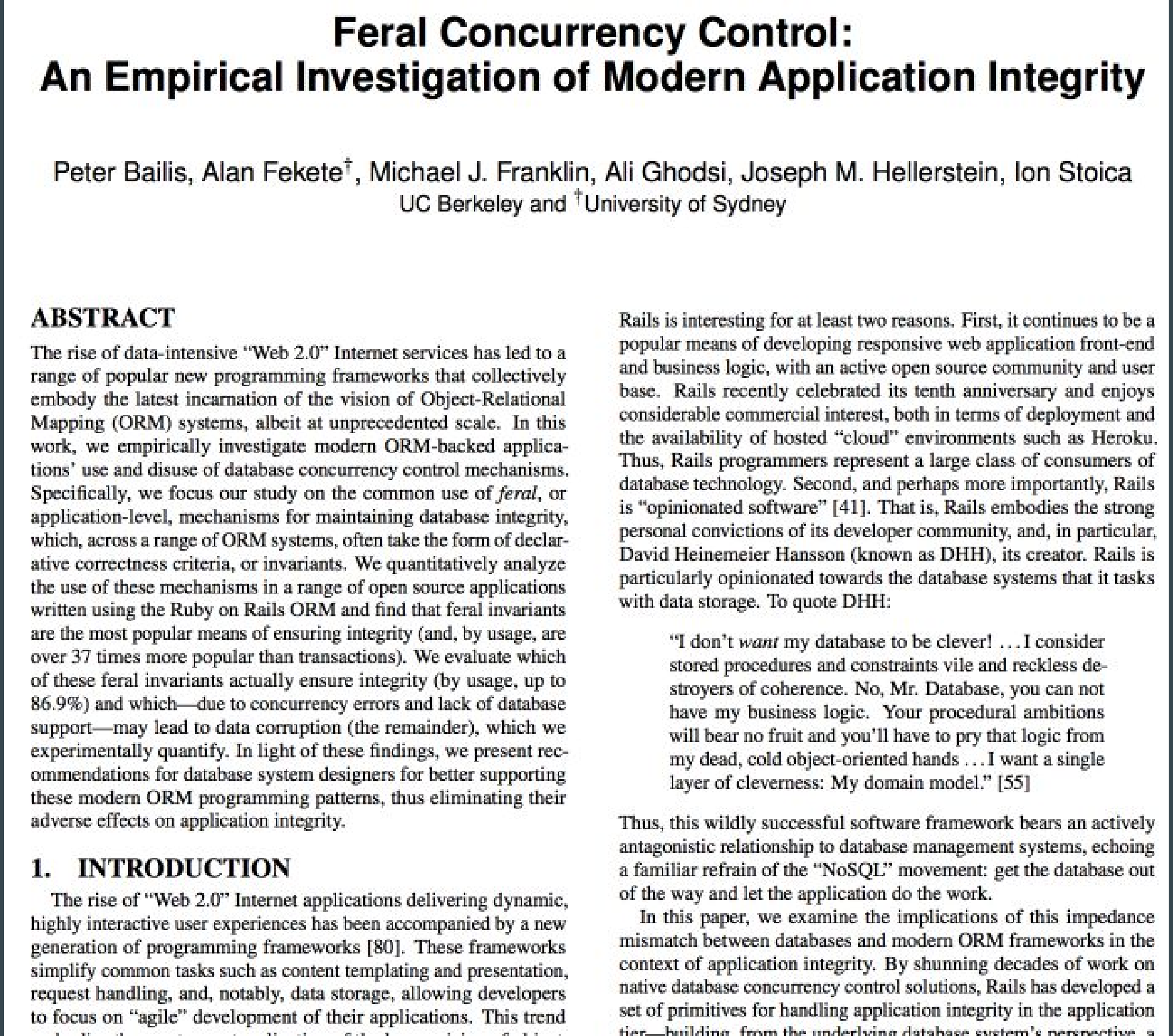
The motivation behind this apparent irrationality

Enhance maintainability of the system

- The set of available integrity and concurrency control mechanisms at the database layer depends on the data model employed by the database
- (e.g: PostgresSQL) support foreign key constraints, while some other (e.g: MySQL's MyISAM and NDB storage engines) do not.
- the CHECK constraint used to check domain-specific integrity constraints, is supported by PostgresSQL, but not supported by MySQL

Facilitate testing

- In many cases, developers want to simply *dump* the sample data (known to be consistent) into the database and get along with testing their application.



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

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

BEGIN/SELECT/INSERT/COMMIT is not safe at READ COMMITTED

…

Most databases default to READ COMMITTED

Why do ORM authors prefer the slow way?

•••

One vs Four Queries !

The right way

Requiers O(n) error handling

```
try
  return User.create(:email =>
$1)
rescue SqliteError { ... }
rescue MysqlError { ... }
rescue PostgresError { ... }
```

The Rails way

DataBase agnostic

```
user = User.find.where(:email =>
$1)

if user?

  throw DuplicateError('User with
that key already exists')

User.create(:email => $1)

tx.commit()
```

Another bad ORM pattern

•••

Read - Check - Update - Save

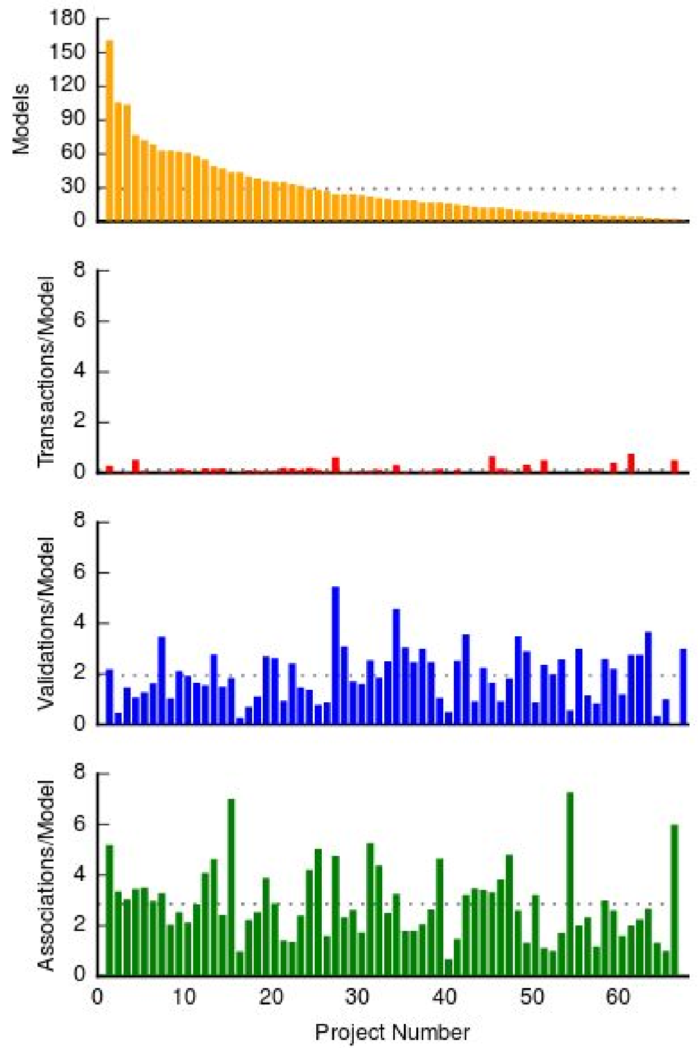
Article suggestion : Don’t use `save()`

```
Cars.update().where(
    :id => 'car_1',
    :state => 'FREE',
).set(
    :state => 'ASSIGNED',
    :driver_id => 'drv_1',
)
# Check for 0 or 1 rows
```

Back to the paper

Article Results

- 67 Rails project surveyed
- used just 3.8 transactions against 52.4 validations and 92.8 associations
- Most consistency checks are in the application



Learn to evaluate ORM’s

- What SQL does User.create() run?
- Check FK behavior
- Check error handling on constraint failures

Use ORM’s better

- Use database constraints
- UPDATE with a WHERE clause, not .save()
- Give up on same ORM, different database

Conclusion

DATABASE EXPERTS:
(do 30 years of research on consistency and locking)

RAILS: Nah, we're good

What can be done about the anomalies?

The paper concludes that there is currently no database-backed solution that “respects and assists with application programmer desires for a clean way of expressing correctness criteria in domain logic

Authors believe that “there is an opportunity and pressing need to build systems that provide all three criteria: performance, correctness, and programmability.”

Application users and framework authors need a new database interface that will enable them to

- Cross-DB flexibility
- Common interfaces for error handling, CRUD
- Make it easier for ORM developers to use your database

Thanks !

•••

These slides are available at
github.com/aribyassine/slides