Ruby on Rails

Open Academy
Spring 2015

Hyonjee Joo, Ben Kogan, James Wen

Overview

- What is Ruby on Rails?
- Small features and fixes
    - `:time` option for `#touch`
    - rake restart
- Main project - Rescuing database errors
    - the problem, our solution, demo, and benchmarking
- Rails Ecosystem
- Takeaways

What is Ruby on Rails?

- Open Source Web Application Development Framework
- Full Stack
- MVC (Model-View-Controller)

Why Popular?

- Quick Start
- CoC (Convention over Configuration) + Structuring
- Solid open source community
- Popular usage:

- Github	- Groupon	- Indiegogo
- Airbnb	- Hulu	- Shopify
- Square	- Soundcloud	- Pivotal
- Basecamp	- Crunchbase	- Scribd

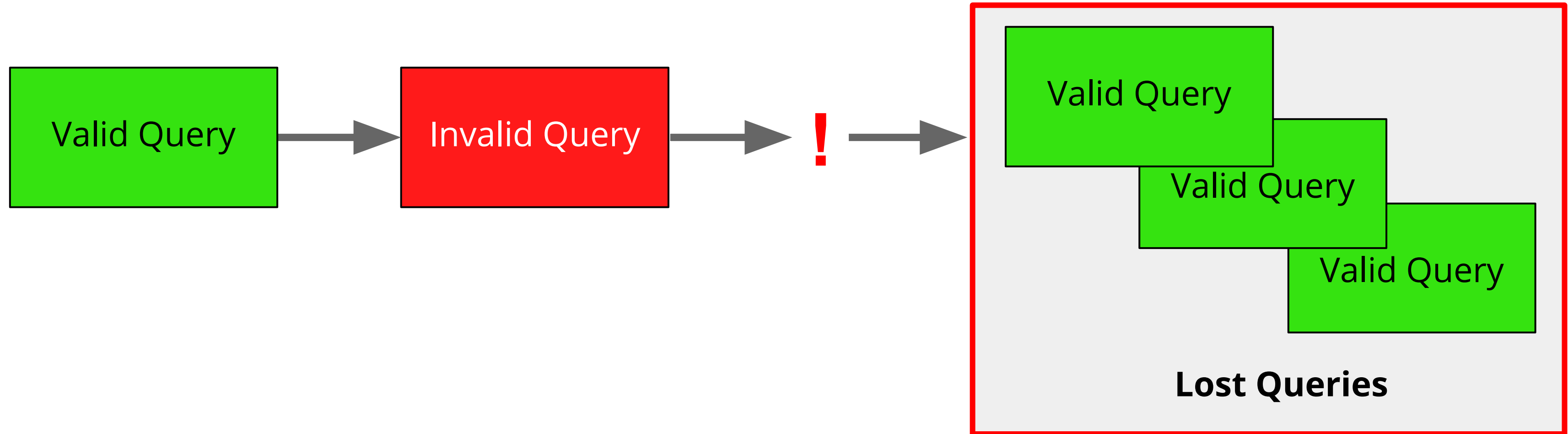
`:time` option for `#touch`

- activerecord `#touch` method
  - saves record with updated_at/on attributes set to current time or time specified by optional `:time` parameter
- avoids having to manually change model attribute if anything besides current time is desired
- example uses: to match two records, set a specific transaction time, alter modify time, etc.

rake restart

- wraps `touch tmp/restart.txt` into a rake task that can be executed on the command line
- signals to rails application server to restart
- useful when implementing small changes
- avoids manual restart

Project: Rescuing Database Errors



Project: Rescuing Database Errors

- Example of the existing problem:

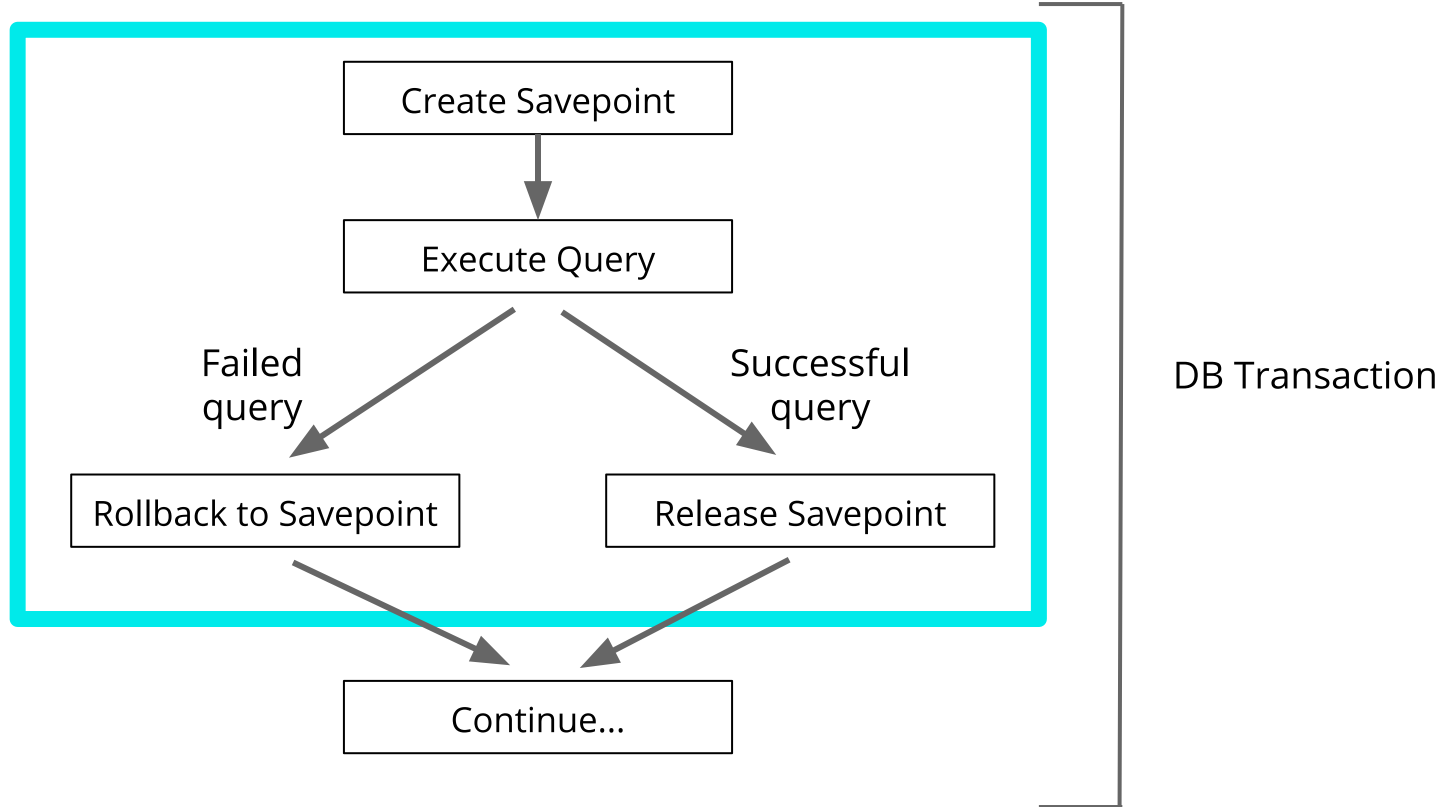
```ruby
# Suppose that we have a Number model with a unique column called 'i'.
Number.transaction do
  Number.create(i: 0)
  begin
    # This will raise a unique constraint error...
    Number.create(i: 0)
  rescue ActiveRecord::StatementInvalid
    # ...which we ignore.
  end

  # On PostgreSQL, the transaction is now unusable. The following
  # statement will cause a PostgreSQL error, even though the unique
  # constraint is no longer violated:
  Number.create(i: 1)
  # => "PGError: ERROR: current transaction is aborted, commands
  #    ignoreed until end of transaction block
end
```

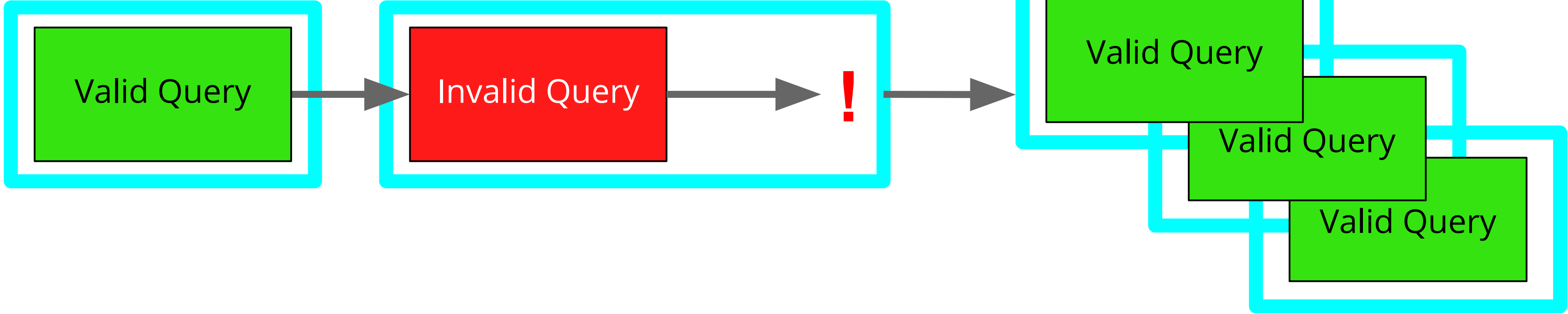
Demo

Solution

Savepoint Wrapping



Solution



DB Transaction

```ruby
def protected_query
  sp_name = 'query_savepoint'
  trans_init = @connection.transaction_status
  in_valid_trans_init = (trans_init == 1 || trans_init == 2)
  if in_valid_trans_init
    create_savepoint sp_name
  end

  begin
    result = yield
  rescue => error
    #Database command error, do rollback
    if @connection.transaction_status != 0
      exec_rollback_to_savepoint sp_name
    end
    raise error
  end

  trans_final = @connection.transaction_status
  in_valid_trans_final = (trans_final == 1 || trans_final == 2)
  if in_valid_trans_init && in_valid_trans_final
    release_savepoint sp_name
  end
  result
end

# Queries the database and returns the results in an Array-like object
def query(sql, name = nil) #:nodoc:
  log(sql, name) do
    protected_query do
      result_as_array @connection.async_exec(sql)
    end
  end
end
```

Benchmarking

- General Framework: ActionDispatch::Performance Test
- Performance of Operations: benchmark-ips gem
- Branches: master vs. postgres-query-savepoints
- Within Test: Regular Queries vs. Transactions
- Run: rake test:benchmark
- Iterations/100 ms & Iteration/s

Benchmarking Code

```ruby
1 class PostgresSavepointTest < ActionDispatch::PerformanceTest
2   # Refer to the documentation for all available options
3   self.profile_options = { runs: 1, metrics: [:wall_time, :memory, :process_time, :cpu_time]}
4
5   test "postgres-savepoints" do
6     name_id_hash = {}
7     Benchmark.ips do |x|
8       x.report("no_transaction_operations") { no_trans_op }
9       x.report("transaction_operations") { trans_op }
10    end
11  end
12
13  def no_trans_op
14    song = Song.new(name: "test", duration: 1, genre: "genre")
15    song.save
16    id = song.id
17    song = Song.find_by(id: id)
18    song.destroy
19  end
20
21  def trans_op
22    Song.transaction do
23      song = Song.new(name: "test", duration: 1, genre: "genre")
24      song.save
25      id = song.id
26      song = Song.find_by(id: id)
27      song.destroy
28    end
29  end
30 end
```

Benchmarking Results

Without Savepoints:

```
Calculating
-------------------------------------
no_transaction_operations
       40.000  i/100ms
transaction_operations
       42.000  i/100ms
-------------------------------------
no_transaction_operations
       409.431  (± 8.8%) i/s -      2.050k
transaction_operations
       429.502  (± 9.5%) i/s -      2.150k
```

With Savepoints:

```
Calculating
-------------------------------------
no_transaction_operations
       52.000  i/100ms
transaction_operations
       56.000  i/100ms
-------------------------------------
no_transaction_operations
       517.106  (±10.4%) i/s -      2.600k
transaction_operations
       553.855  (± 7.8%) i/s -      2.800k
```

Benchmarking Analysis/Thoughts

- No transaction: (517.106 - 409.431) / 409.431 = .262986926 * 100 = 26.3% slower
- Transaction: (553.855 - 429.502) / 429.502 = .289528337 * 100 = 29.0% slower
- Back of the Envelope calculations
- Quick Github Search: 394,198 instances of (gem ‘pg’) in repos.
- Quick Github Search: 86,359 instances of (gem ‘mysql’) in repos.
- Quick Github Search: 672,995 instances of (gem ‘sqlite3’) in repos.
- 34% of Rails users/apps use postgres.
- 34% of Rails apps will suffer
- Acceptable? Not acceptable?
- Experiment: Cut down speed decrease (@connection.transaction_status)
- Considerations: How many production Rails apps/instances use postgres? How often are transactions used? How often do Rails apps that use postgres use transactions?

Why PostgreSQL?

- only PostgreSQL blocks after erroring queries within a transaction

MySQL

```
mysql> start transaction;
Query OK
mysql> select * from fu;
ERROR: Table 'fu' doesn't exist
mysql> select * from fun;
+--------+
| c      |
+--------+
| hello! |
+--------+
1 row in set (0.00 sec)
mysql> commit;
Query OK
```

PostgreSQL

```
mydb=# BEGIN;
BEGIN
mydb=# SELECT * FROM fu;
ERROR: relation "fu" does not exist
mydb=# SELECT * FROM fun;
ERROR: current transaction is
aborted, commands ignored until end
of transaction block
mydb=# END;
ROLLBACK
```

SQLite

```
sqlite> begin;
sqlite> select * from fu;
Error: no such table: fu
sqlite> select * from fun;
hello!
sqlite> commit;
```

Challenges

- Wading our way through poorly documented code
- Working with Postgres through the PG gem
  - determining how deep down into the stack we can determine the transaction status
  - selecting the appropriate synchronous or asynchronous exec methods for sending queries
- Integrating the new savepoint feature seamlessly without breaking other expected behavior
  - much time spent modifying our solution to pass all previously existing tests in the activerecord test suite
- Learning Ruby
- Getting familiar with the open source community

Ruby/Rails Ecosystem

- Rails and Ruby Gems
- Easy to try to contribute, tougher to actually contribute
- Liked: Helpful changes, reproducibility, benchmarking, convention
- Tech companies that use it a lot also contribute a lot (expected)
- Issues + Pull Requests → Rails Core Team Review + Advise + Merge

Get Started/Involved:

- Mailing List
- Google Groups
- **Github Repo Watch**
- Programs like this (OpenAcademy), Google Summer of Code, etc.

Takeaways

- It’s not easy working with a huge code base
- It’s important to adhere to the development process
  - squash commits
  - add changes to CHANGELOG.md
  - document new features or fixes
- Ask questions, then ask more questions, and then more questions
- Open source isn’t always pretty
- You can contribute without knowing the entire picture

Thanks to Professor Jae, Professor Cannon, and our Rails mentors!

Eileen Uchitelle, Matthew Draper, Aaron Patterson, Andrew White, Jeremy Kemper