

---

## PostgreSQLCursor for handling large Result Sets

gem version 0.6.9

PostgreSQLCursor extends ActiveRecord to allow for efficient processing of queries returning a large number of rows, and allows you to sort your result set.

In PostgreSQL, a cursor runs a query, from which you fetch a block of (say 1000) rows, process them, and continue fetching until the result set is exhausted. By fetching a smaller chunk of data, this reduces the amount of memory your application uses and prevents the potential crash of running out of memory.

Supports Rails/ActiveRecord v3.1 (v3.2 recommended) higher (including v5.0) and Ruby 1.9 and higher. Not all features work in ActiveRecord v3.1. Support for this gem will only be for officially supported versions of ActiveRecord and Ruby; others can try older versions of the gem.

### Using Cursors

PostgreSQLCursor was developed to take advantage of PostgreSQL's cursors. Cursors allow the program to declare a cursor to run a given query returning "chunks" of rows to the application program while retaining the position of the full result set in the database. This overcomes all the disadvantages of using `find_each` and `find_in_batches`.

Also, with PostgreSQL, you have an option to have raw hashes of the row returned instead of the instantiated models. An informal benchmark showed that returning instances is a factor of 4 times slower than returning hashes. If you can work with the data in this form, you will find better performance.

With PostgreSQL, you can work with cursors as follows:

```
1 Product.where("id>0").order("name").each_row { |hash| Product.process(hash) }
2
3 Product.where("id>0").each_instance { |product| product.process! }
4 Product.where("id>0").each_instance(block_size:100_000) { |product|
  product.process }
5
6 Product.each_row { |hash| Product.process(hash) }
7 Product.each_instance { |product| product.process }
8
9 Product.each_row_by_sql("select * from products") { |hash| Product.process(hash) }
10 Product.each_instance_by_sql("select * from products") { |product|
  product.process }
```

---

Cursors must be run in a transaction if you need to fetch each row yourself

```
1 Product.transaction do
2   cursor = Product.all.each_row
3   row = cursor.fetch           #=> {"id"=>"1"}
4   row = cursor.fetch(symbolize_keys:true) #=> {:id =>"2"}
5   cursor.close
6 end
```

All these methods take an options hash to control things more:

```
1 block_size:n      The number of rows to fetch from the database each
   time (default 1000)
2 while:value       Continue looping as long as the block returns this
   value
3 until:value       Continue looping until the block returns this value
4 connection:conn   Use this connection instead of the current Product
   connection
5 fraction:float     A value to set for the cursor_tuple_fraction variable
   .
6                   PostgreSQL uses 0.1 (optimize for 10% of result set)
7                   This library uses 1.0 (Optimize for 100% of the
   result set)
8                   Do not override this value unless you understand it.
9 with_hold:boolean Keep the cursor "open" even after a commit.
10 cursor_name:string Give your cursor a name.
```

Notes:

- Use cursors *only* for large result sets. They have more overhead with the database than ActiveRecord selecting all matching records.
- Aliases each\_hash and each\_hash\_by\_sql are provided for each\_row and each\_row\_by\_sql if you prefer to express what types are being returned.

## PostgreSQLCursor is an Enumerable

If you do not pass in a block, the cursor is returned, which mixes in the Enumerable library. With that, you can pass it around, or chain in the awesome enumerable things like `map` and `reduce`. Furthermore, the cursors already act as `lazy`, but you can also chain in `lazy` when you want to keep the memory footprint small for rest of the process.

```
1 Product.each_row.map {|r| r["id"].to_i } #=> [1, 2, 3, ...]
2 Product.each_instance.map {|r| r.id }.each {|id| p id } #=> [1, 2, 3,
   ...]
3 Product.each_instance.lazy.inject(0) {|sum,r| sum + r.quantity } #=>
   499500
```

---

## Hashes vs. Instances

The `each_row` method returns the Hash of strings for speed (as this allows you to process a lot of rows). Hashes are returned with String values, and you must take care of any type conversion.

When you use `each_instance`, ActiveRecord lazily casts these strings into Ruby types (Time, Fixnum, etc.) only when you read the attribute.

If you find you need the types cast for your attributes, consider using `each_instance` instead. ActiveRecord's read casting algorithm will only cast the values you need and has become more efficient over time.

## Select and Pluck

To limit the columns returned to just those you need, use `.select(:id, :name)` query method.

```
1 Product.select(:id, :name).each_row { |product| product.process }
```

Pluck is a great alternative instead of using a cursor. It does not instantiate the row, and builds an array of result values, and translates the values into ruby values (numbers, Timestamps. etc.). Using the cursor would still allow you to lazy load them in batches for very large sets.

You can also use the `pluck_rows` or `pluck_instances` if the results won't eat up too much memory.

```
1 Product.newly_arrived.pluck(:id) #=> [1, 2, 3, ...]
2 Product.newly_arrived.each_row { |hash| }
3 Product.select(:id).each_row.map {|r| r["id"].to_i } # cursor instead
  of pluck
4 Product.pluck_rows(:id) #=> ["1", "2", ...]
5 Product.pluck_instances(:id, :quantity) #=> [[1, 503], [2, 932], ...]
```

## Associations and Eager Loading

ActiveRecord performs some magic when eager-loading associated row. It will usually not join the tables, and prefers to load the data in separate queries.

This library hooks onto the `to_sql` feature of the query builder. As a result, it can't do the join if ActiveRecord decided not to join, nor can it construct the association objects eagerly.

---

## Locking and Updating Each Row (FOR UPDATE Queries)

When you use the AREL `lock` method, a “FOR UPDATE” clause is added to the query. This causes the block of rows returned from each FETCH operation (see the `block_size` option) to be locked for you to update. The lock is released on those rows once the block is exhausted and the next FETCH or CLOSE statement is executed.

This example will run through a large table and potentially update each row, locking only a set of rows at a time to allow concurrent use.

```
1 Product.lock.each_instance(block_size:100) do |p|
2   p.update(price: p.price * 1.05)
3 end
```

Also, pay attention to the `block_size` you request. Locking large blocks of rows for an extended time can cause deadlocks or other performance issues in your application. On a busy table, or if the processing of each row consumes a lot of time or resources, try a `block_size <= 10`.

See the PostgreSQL Select Documentation for more information and limitations when using “FOR UPDATE” locking.

## Background: Why PostgreSQL Cursors?

ActiveRecord is designed and optimized for web performance. In a web transaction, only a “page” of around 20 rows is returned to the user. When you do this

```
1 Product.where("id>0").each { |product| product.process }
```

The database returns all matching result set rows to ActiveRecord, which instantiates each row with the data returned. This function returns an array of all these rows to the caller.

Asynchronous, Background, or Offline processing may require processing a large amount of data. When there is a very large number of rows, this requires a lot more memory to hold the data. Ruby does not return that memory after processing the array, and the causes your process to “bloat”. If you don’t have enough memory, it will cause an exception.

## ActiveRecord.find\_each and find\_in\_batches

To solve this problem, ActiveRecord gives us two alternative methods that work in “chunks” of your data:

```
1 Product.where("id>0").find_each { |model| Product.process }
2
```

---

```
3 Product.where("id>0").find_in_batches do |batch|
4   batch.each { |model| Product.process }
5 end
```

Optionally, you can specify a `:batch_size` option as the size of the “chunk”, and defaults to 1000.

There are drawbacks with these methods:

- You cannot specify the order, it will be ordered by the primary key (usually id)
- The primary key must be numeric
- The query is rerun for each chunk (1000 rows), starting at the next id sequence.
- You cannot use overly complex queries as that will be rerun and incur more overhead.

## How it works

Under the covers, the library calls the PostgreSQL cursor operations with the pseudo-code:

```
1 SET cursor_tuple_fraction TO 1.0;
2 DECLARE cursor_1 CURSOR WITH HOLD FOR select * from widgets;
3 loop
4   rows = FETCH 100 FROM cursor_1;
5   rows.each {|row| yield row}
6 until rows.size < 100;
7 CLOSE cursor_1;
```

## Meta

### Author

Allen Fair, @allenfair, [github://afair](https://github.com/afair)

### Note on Patches/Pull Requests

- Fork the project.
- Make your feature addition or bug fix.
- Add tests for it. This is important so I don’t break it in a future version unintentionally.
- Commit, do not mess with rakefile, version, or history. (if you want to have your own version, that is fine but bump version in a commit by itself I can ignore when I pull)
- Send me a pull request. Bonus points for topic branches.

---

## **Code of Conduct**

This project adheres to the Open Code of Conduct. By participating, you are expected to honor this code.

## **Copyright**

Copyright (c) 2010-2017 Allen Fair. See (MIT) LICENSE for details.