

Deixe o Postgres até 40% mais rápido em seu ActiveRecord

Últimamente notei que o postgres 8.3 estava superando o MySQL em meus testes de performance. Então resolvi testar os dois com ActiveRecord. Nesse caso o Postgres perdia para o MySQL então fui procurar onde estava o problema .

Eu já estava incomodado com o MySQL não suportar transação em alter tables e não possuir alter tables on-line (alter table on-line é quando ela é feita instânea, sem necessidade de cópia temporária da tabela). Isso atrapalha quando você tem migrations que falham durante a execução. Além disso, o fato das alterações de alter table não serem on-line pioram conforme os dados do seu cliente ficam maiores.

Sempre fiz testes comparando o MySQL x PostgreSQL x Firebird. Como todos os meus testes eram na versão 7 do Postgres (já faz algum tempo !) resolvi verificar no google se achava alguma coisa sobre a performance do Postgres série 8.

Localizei no change log do Postgres 8.2 que havia melhoria na performance das queries e alguns artigos mostravam que a versão 8.3 era superior a 8.2, então resolvi compilar e testar.

Ao iniciar os testes fiquei espantado ao ver pela primeira vez o PostgreSQL bater o MySQL em velocidade.

Como os testes que tinha prontos eram em PHP, resolvi testar os dois no ActiveRecord. Fiz questão de efetuar um teste real, com os dados de nossos clientes. Então serializei os objetos ActiveRecord da seguinte forma:

```
File.open('produtos.yml', 'w+')
```

Com os objetos serializados era hora de exportar para o postgres então reconfirei meu projeto Rails para o postgres rodei um rake db:migrate e populei o projeto da seguinte forma

```
Produto::Base #essa linha e necessaria para o rails carregar a classe
YAML::load_file('produtos.yml').each{|obj| obj.send(:create) }
```

Nota: no rails 1.2.6 ele deixa usar o create direto, já no rails 2.0.2 não. E não posso usar save neste caso. Pronto ! Os dois bancos de dados estavam com dados idênticos e reais para testes.

Minha decepção foi rodar os testes e ver que o PostgreSQL estava perdendo em consultas grandes e ganhava do MySQL em consultas pequenas. Também notei que, sendo menor o número de COLUNAS, menor a diferença ficava.

Então refiz os testes utilizando os driver's nativos

```
mysql = MySQL.new(...)
20.times do
  init = Time.now
  mysql.query('SELECT ...')
  puts "resultado do teste: #{Time.now - init}"
end
```

```
conn = PGconn.connect(...)
20.times do
  init = Time.now
  conn.exec('SELECT ...')
  puts "resultado do teste: #{Time.now - init}"
end
```

Nesse caso, os testes representaram a mesma proporção do PHP (5.2.2 utilizando PDO). mas os testes foram ligeiramente mais rápidos que no PHP.

Desconfiando que o problema estava no ActiveRecord fui olhar a implementação do Adapter do Postgres e MySQL e comparei os dois.

No meu caso se encontram em:

```
/usr/local/ruby/lib/ruby/gems/1.8/gems/activerecord-1.15.6/lib/active_record/connection_adapters/
```

```
postgresql_adapter.rb
```

```
mysql_adapter.rb
```

O método responsável por fazer as chamadas era select

Verifique a implementação dos dois:

Mysql:

```
def select(sql, name = nil)
  connection.query_with_result = true
  result = execute(sql, name)
  rows = result.all_hashes
```

```

    result.free
    rows
end

def all_hashes
  rows = []
  each_hash { |row| rows }
end

```

PostgreSQL:

```

def select(sql, name = nil)
  res = execute(sql, name)
  results = res.result
  rows = []
  if results.length > 0
    init = Time.now
    results.each do |row|
      fields = res.fields
      hashed_row = {}
      row.each_index do |cel_index|
        column = row[cel_index]

        case res.type(cel_index)
        when BYTEA_COLUMN_TYPE_OID
          column = unescape_bytea(column)
        when TIMESTAMPTZOID, TIMESTAMPOID
          column = cast_to_time(column)
        when NUMERIC_COLUMN_TYPE_OID
          column = column.to_d if column.respond_to?(:to_d)
        end

        hashed_row[fields[cel_index]] = column
      end
    end
  end
  rows
end

```

Ai ficou fácil verificar que o adapter do postgres faz um tratamento de dados efetuando cast de binary, BidDecimal e Time. Esse tratamento é precoce demais porque o próprio objeto Active Record se encarrega dele no que diz respeito a Time e BigDecimal. Já o cast de dados binários (bytea) teria que ser feito depois. Quando fiz os testes com o ActiveRecord o Postgres estava dando 2.53 e o MySQL 1.95 a 2 (segundos).

Então alterei o código para o seguinte:

```

def select(sql, name = nil)
  res = execute(sql, name)
  results = res.result
  rows = []
  if results.length > 0
    init = Time.now
    results.each do |row|
      rows }
    end
  end
  res.clear
  return rows
end

```

A implementação acima é uma imitação do Adapter do MySQL, achei o método to_hash, que retorna uma hash, e sem o cast "prematuro" que ele faz o tempo que era de 2.53 caiu para 1,73, batendo o MySQL que era de 1.95 a 2.00 (segundos) O cast para BidDecimal e Time acontece do mesmo jeito quando você invoca o método de uma instância de ActiveRecord, exemplo:

```

Produto::Base.find(all).each { |o| o.preco }

```

o.preco efetua um cast para BigDecimal e se não for necessário a chamada do método o overhead do cast é literalmente peso morto !!! Não satisfeito pela minha sede de performance resolvi criar dentro do driver do postgres um método que retornasse toda a coleção. Chamei o método de hash_all e meu adapter ficou assim:

```

def select(sql, name = nil)
  res = execute(sql, name)

```

```

results = res.result
rows = res.hash_all
res.clear
return rows
end

```

Para criar meu código em C abri o arquivo postgres.c e adicionando o seguinte:

```

static VALUE
pgresult_hash_all(self)
  VALUE self;
{
  PGresult *result = get_pgresult(self);
  int row_count = PQntuples(result);
  VALUE fields = pgresult_fields(self);
  VALUE ary = rb_ary_new();
  int row_num;
  int key_num = RARRAY(fields)->len;
  for (row_num = 0; row_num < row_count; row_num++)
    VALUE hash = rb_hash_new();
  int i;
  for (i = 0; i < key_num; i++)
    VALUE key = rb_ary_entry(fields, i);
    VALUE value = fetch_pgresult(result, row_num, i);
    rb_hash_aset(hash, key, value);
  }
  rb_ary_push(ary, hash);
}
return ary;
}

```

e também tive que adicionar a seguinte linha:

```
rb_define_method(rb_cPGresult, "hash_all", pgresult_hash_all, 0);
```

Essa última linha define o nome do método que será visto em objetos ruby. Outras coisas interessantes nesse código são:

rb_ary_new; cria um array ruby
rb_ary_push; adiciona um elemento a um array
rb_hash_new; cria uma hash ruby
rb_hash_aset; adiciona um elemento a hash

Não é tão difícil assim né ?!

Bom rodei os testes de novo e a mudança não foi tão espantosa, mas nesse caso o tempo chegou até 1.63, algo em torno do 5% ou 6%.

Então rodei o mesmo teste só que usando várias thread's simultâneas e a diferença chegou a 12 % !!!!

Não vou usar essa solução em produção mas algo me chamou a atenção, que tal um Adapter escrito totalmente em 'C'. Seria pelo menos divertido de escrever !!!!

Agora para algo mais prático vamos criar um plugin que altere o Rails e implemente nosso novo método mais rápido , lembrando que se você precisar de cast bytea faça-o posteriormente.

Crie um plugin em seu projeto da seguinte forma:

script/generate plugin pgsqldb_improved
no arquivo init.rb escreva o seguinte:

```

class ActiveRecord::ConnectionAdapters::PostgreSQLAdapter
  def select(sql, name = nil)
    res = execute(sql, name)
    results = res.result
    rows = []
    if results.length > 0
      results.each do |row|
        rows << row
      end
    end
  end
end

```

E só !!! Nosso plugin está pronto !!!

Se voce quiser baixar o driver em 'C' que eu alterei clique aqui:

driver

Se voce quiser baixar o plugin acima clique aqui:

plugin

Bom... meus testes com o postgresql apenas começaram, mas acho que isso ajuda quem já meche com ele .

Abraço, a todos

Alexandre Riveira