Relational Data Handling

Siddharth Kaza

Towson University, Computer and Information Sciences

Outline

- Web Applications and Relational Data
  - Introduction
  - Simple CRUD operations in ActiveRecord (Rail's ORM)

- Relationships between models/tables
- STI and Polymorphic associations
- Transactions

Relational data

- Relational data – model relations between items
  - Containment – one:many
    - cart contains many line_items
  - Relationships – many:many
    - Authors have many books
    - Books have many authors
- How do we do this?

Traditional approach

1. Model things in SQL

2. Every table has a primary key
    1. Identifier that uniquely names each row
    2. Try to maintain normal forms
    3. Include foreign keys in a table to indicate a relationship

3. Use SQL queries to create associations
4. Write code to process rows in query result.

1. Model in SQL

- Consider a book store
- Table definitions with primary keys

Primary keys

```
create table books (book_id int, title varchar(80), publisher_id int)
create table publishers (publisher_id int, name varchar(80))
create table authors (author_id,int, first_name varchar(20),
                    last_name varchar(30))
create table authors_books (author_id int, book_id int)
```

Foreign keys

- Normal forms
  - Each column is atomic
  - Other columns must provide facts about the primary key and nothing but the primary key

2. SQL Query and 4. processing result

-
  ```
  query = “Select b.title, p.name, a.first_name, a.last_name  from
  book b, publishers p, authors a, authors_books ab
  where b.publisher_id=p.publisher_id
          and ab.book_id=b.book_id
          and ab.authors_id = a.authors_id
          and b.book_id=36”;
  ```

-
  ```
  ResultSet res = stmt.executeQuery(query)
  while (res.next) {
          String booktitle = rs.getString(1);
          String pubname = rs.getString(2);
          …
  ```

- stuff things into variables, classes, objects, etc.

What's the problem with this?

- SQL is painful to write
  - inside java/C#/… code, DB enthusiasts like to write stored procedures outside
- Follow-on code must be modified if query changes
- Am I pulling in too much data? Too little?
- Relationships are not easily discernible from code
- Details in too many places
- => Need for Object-Relational Mapping

CRUD

- ORM should be able to provide the basic DB operations

- Create
- Read
- Update
- Delete

- Let look at these from the depot context

- Find good details here: http://guides.rubyonrails.org/active_record_querying.html

Create

- New creates new object

- Save saves it

```
an_order = Order.new
an_order.name = "Dave Thomas"
an_order.email = "dave@pragprog.com"
an_order.address = "123 Main St"
an_order.pay_type = "check"
an_order.save
```

More create

- Use hash (useful to get data from forms)

```
an_order = Order.new(
    :name => "Dave Thomas" ,
    :email => "dave@pragprog.com" ,
    :address => "123 Main St" ,
    :pay_type => "check" )
an_order.save
```

- use block

```
user = User.new do |u|
    u.name = "David"
    u.occupation = "Code Artist"
end
```

save vs. save!, create vs. create!

- save returns true if record saved, nil otherwise
  - Nil if validation errors
  - Assume coming from a form that will show errors

- create returns object whether saved or not
  - You must check the validations

- save!, create! Raise exception
  - Use this to force you to write code that handles errors

```
if order. save
    # all OK
else
    # validation failed
end
```

READ - Find

p = Product.find(1)

- Every model class supports the find method
- Takes one or more primary keys
- If given one
  - Returns object for that row
- If multiple – returns array
- RecordNotFound if any not found
- Product.find(27)
  - Select * from product where product_id=27

  Product.find([27,34,59])
  - Select * from product where product_id in (27,34,59);

Find all, first

- Product.all
  - Finds all
- Product.first
  - Finds first
  - Same query, but *first* just returns one row
- Subject to conditions
  - pos = Order.where("name = 'Dave' and pay_type = 'po'")
- Essentially generates arbitrary conditions in SQL
  - The string after condition is just appended to the generated statement

Warning on using :conditions

- Parameterize them – not a hard-wired query

```
pos = Order.where("name = '#{name}' and pay_type =
    'po'" )
```

- Not a good idea! – if name is entered from a form

- SQL injection – can open up security holes
  - consider this: the user enters “john OR 1--“ in the name field
  - “OR 1” will make everything true
  - “--“ will comment everything after that

Warning on using :conditions (cont.)

- Use ? as a placeholder

```
pos = Order.where ("name = ? and pay_type = 'po'" ,
name)
```

- Or hash (with named parameters)

```
pos = Order.where("name = :name and pay_type = :pay_type"
  ,{:pay_type => pay_type, :name => name})
```

- When you use parameters - '?', Active Record will quote all characters that have special meaning for that DB adapter.

Warning on using :conditions (cont.)

- If you need to execute query with similar options several places in your code
  - then put it as a method in the model

Instead of

```
Email.where(“user_id = ? and read=‘no’”,
               user.id)
```

Do this

```
  class Email < ActiveRecordBase
     def self.find_unread_email_for_user (user)
                 Email.where(“user_id = ? and
                              read=‘no’”, user.id)
     end
end
```

Send params for query as a hash

- Suppose form asks for name and pay type in an Order object

```
pos = Order.where("name = :name and pay_type = :pay_type" ,
                                          params[:order]])
```

- Or

```
pos = Order.where(params[:order])
```

  - This will search on all fields in form – not just name and pay_type
    - It will use the 'and' operator between conditions in the where clause

Other find options

- :order
- : limit
- :offset
  - (for pagination, etc.)
- :joins
  - Model should cover this for you
- :select
  - Which columns – not just all
- :readonly
- :group
- :lock
  - Lock the row while you are working on it, DB specific

Using native SQl in find

- find_by_sql

```
orders = LineItem.find_by_sql
("select line_items.* “+
“ from line_items, orders " +
" where order_id = orders.id " +
" and orders.name = 'Dave Thomas' " )
```

More finds – Aggregates

- Order.average(:amount)
- Order.maximum(:amount)
- Order.minimum(:amount)
- Order.sum(:amount)

- std_dev = Order.calculate(:std ,:amount)
  - std as an SQL function
  - Options similar to others
  - Can be grouped

Grouping and finds

- result = Order.maximum :amount, :group => "state"

puts result #=> [["TX", 12345], ["NC", 3456], ...]

Counts

- No parameters – all rows
  - Order.count

- With parameters
- result = Order.count "amount > 10“

- By sql

```
count = LineItem.count_by_sql(
"select count(*) " +
“from line_items, orders " +
“where line_items.order_id = orders.id " +
“and orders.name = 'Dave Thomas' " )
```

Dynamic finders

- Order – attributes name, email, address

```
order = Order.find_by_name("Dave Thomas" )
orders = Order.find_all_by_name("Dave Thomas" )
order = Order.find_all_by_email(params['email' ])
```

- find_by_xxx is :first
- find_all_by_xxx is :all

- Conjunctions (deprecated in Rails 4, gem available)

```
user = User.find_by_name_and_password(name, pw)
```

- Combine create and find – create if not there

```
cart = Cart.find_or_initialize_by_user_id(user.id)
cart.save
```

Update

- Simply modify and save

```
order = Order.find(1)
order.quantity = order.quantity+1
order.save
```

- update_attributes

```
order = Order.find(321)
order.update_attributes(:name => "Barney" , :email =>
"barney@bedrock.com" )
```

- Combine with find
  - `order = Order.update(12, :name => "Barney" , :email => "barney@bedrock.com" )`

Delete

order.destroy

- Deletes row from db
- Freezes object from future changes

order.destroy_all ([“shipped_at < ?”, 30.days.ago])

- Destroy all order object

- The destroy method is a ORM level method
  - It takes into consideration all validations, relationships, and callbacks
- There is another method called delete (order.delete)
  - This is a pure SQL method and not recommended

Relationships

- How do we define them? – foreign keys

- Pros:
  - Forces database to do some work for you
  - Gives us explicit relationships

- Cons:
  - RDMBS-specific
  - May be hard to debug when things go wrong

- Add them (optionally) upon deployment?

Defining Relationships in ORM

- There is the DB level
  - foreign keys

- and then there is the ORM/Rails level.

Using the Depot example from the book:

```
class Order < ActiveRecord::Base
   has_many :line_items

class Product < ActiveRecord::Base
   has_many :line_items
   # ...
end
```

Relationships in Rails Models

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product
end
```

- tells rails that the rows in line_items are children of rows in the orders and products table

- What do these do?

these are methods

```
        li = LineItem.find(...)
        puts "This line item was bought by #{li.order.name} "

        order = Order.find(...)
        puts "This order has #{order.line_items.size} line items"
```

More on ActiveRecord and Relations

- Active record lets us naturally walk relationships in object-oriented manner
- Consider – each line_item references a product
- No need to write SQL to join

```
select      p.price
from        line_items li,
            products p
where       li.product_id = p.id
            and p.id = 36
```

line_item_id	order_id	product_id

- Also, don't need to find second object

```
product_id = line_item.product_id
product = Product.find(product_id)
price = product.price
```

- use relations

```
price = line_item.product.price
```

Primary Keys (aside)

- Primary key is the unique identifier for a database row
- Generally, but not always, an integer
- Rails really wants you to have an integer as a primary key
  - Rails wants to call it id
  - Usually a good idea
  - Even if you have some external id for something, having your own internal version might help
  - You can switch it off however -

```
create_table :categories_products, :id => false do |t|
    t.column :product_id, :integer
    t.column :category_id, :integer
end
```

Primary Keys (cont.)

- ISBNs, SSNs are supposedly unique
  - But may not be…
  - Also what if ISBN changes? – What would you have to do?
  - Why would you not use SSN?

```
class Book < ActiveRecord::Base
    self.primary_key = "isbn"
end

create_table :tickets, :primary_key => :number do |t|
        t.column :created_at, :timestamp
        t.column :description, :text
        t.column :number, :integer
end
```

Naming tables

- Class names get split at caps and pluralized to form table names
  - LineItem class becomes line_items table
  - But, you use rails generate model line_item
- Inherent tradeoff
  - Attempt at "smart" guessing of names simplifies coding, initial installation, etc.
- But, it may not always work the way you want
  - Might not always agree with the designer's decisions
- Alternative – configure it explicitly in files/code
  - Read through the configuration to figure it out
    - set_table_name "name" in model
    - or look at config/initializers/inflections.rb

ActiveRecord – Relationships in detail

- ActiveRecord supports three types of relationships
  - one – many
  - many – many
  - one - one

Setting up relationships

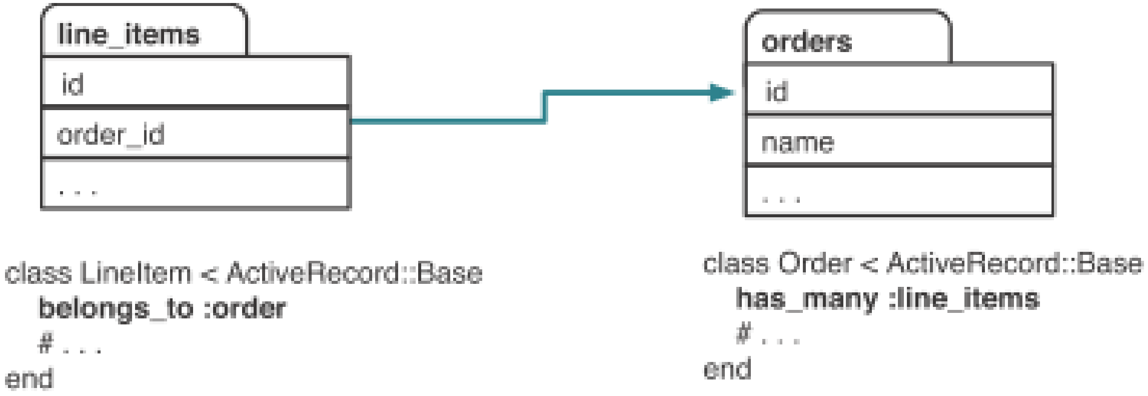
- Two pieces
  - Fields in table
  - Migration for model yyy has xxx_id as a column if each y has a foreign key with x
    - ie., x is associated with many y's
    - Or, x contains many y's

```
class CreateLineItems < ActiveRecord::Migration
    def self.up
            create_table :line_items do |t|
                    t.column :product_id, :integer, :null => false
                    t.column :order_id, :integer, :null => false
                    t.column :quantity, :integer, :null => false
                    t.column :total_price, :decimal, :null => false
        end….
```

Named after
the class not
the table
- singular!

One – Many

- Example – relationship between line_items and orders

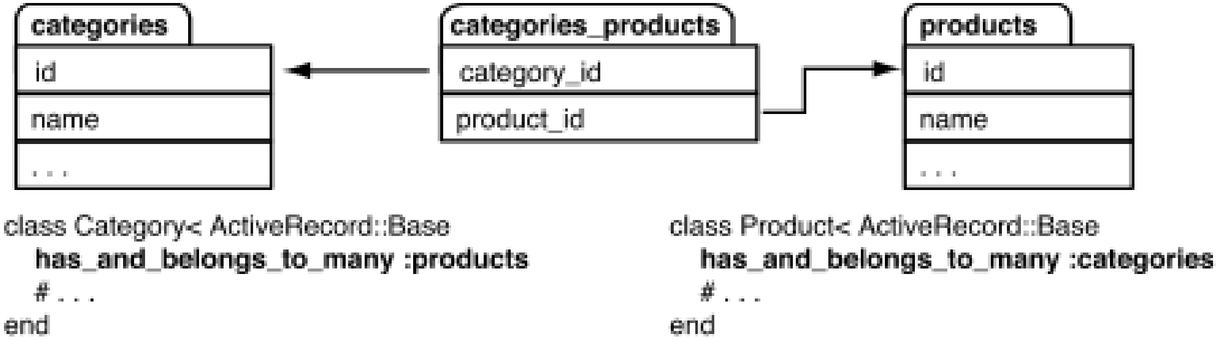


```
class LineItem < ActiveRecord::Base
  belongs_to :order
  # . . .
end
```

```
class Order < ActiveRecord::Base
  has_many :line_items
  # . . .
end
```

Many – Many

- Example – Categories and Products – A product may have many categories and a category many have many products



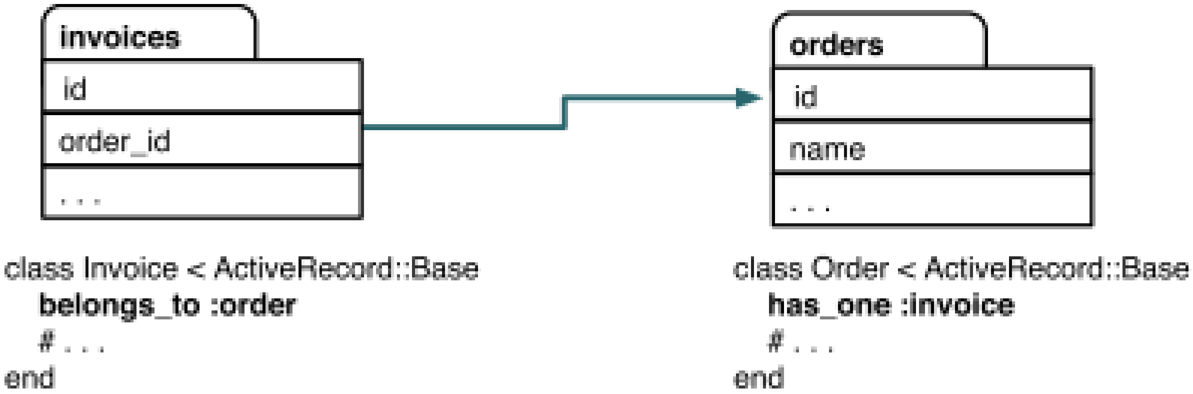
```
class Category< ActiveRecord::Base
  has_and_belongs_to_many :products
  # . . .
end
```

```
class Product< ActiveRecord::Base
  has_and_belongs_to_many :categories
  # . . .
end
```

- Within the db the many to many is represented by a join table – table1_table2 (in alphabetical order)

One - One

- Example – relationship between orders and invoices



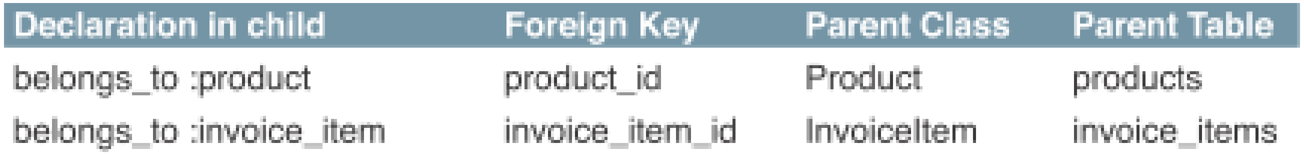
```
class Invoice < ActiveRecord::Base
  belongs_to :order
  # . . .
end
```

```
class Order < ActiveRecord::Base
  has_one :invoice
  # . . .
end
```

- Model with the foreign_key has the belongs_to declaration

Notes on this

- Foreign key named after class of table with _id appended
- Be careful of Rails' cleverness with plurals
  - Person model will map to people table, and person_id will be foreign key
- has_many takes full table name, as in database
- item with foreign key (line_item) always is the belongs_to
  - When you write *belongs_to* think you are writing *references* in SQL terms



Declaration in child	Foreign Key	Parent Class	Parent Table
belongs_to :product	product_id	Product	products
belongs_to :invoice_item	invoice_item_id	InvoiceItem	invoice_items

Methods that clauses provide us

- belongs_to
  - line_item belongs_to :product
    - line_item.product – accessor and mutator
      - □ must save after mutate
  - build_product (attribute hash) coded as
    - line_item.product = Product.new (…..)
      - □ create a new product
      - □ must save it

Dependency options for has_many

- :dependent=>:destroy
  - goes through child row and deletes each that refers to parent being deleted
- :dependent=>:delete_all
  - single sql query to delete all
- can override the search SQL by using the :finder_sql options
- can change the ordering
  - :order => “quantity, unit_price DESC”

Methods you get with has_many

- order has_many line_items
  - line_items – get array
  - line_items << line_item
  - line_items.push(line_item) – add to list
  - line_items.replace(item1,item2...)
    - replace old with new – detects differences and optimizes DB access
  - line_items.delete(item1, items2...)
    - invokes destroy as necessary
  - line_items.delete_all
  - line_items.destroy_all
  - line_items.find
  - size, count, length (forces reload), empty, sum, uniq, build, create

has_one options

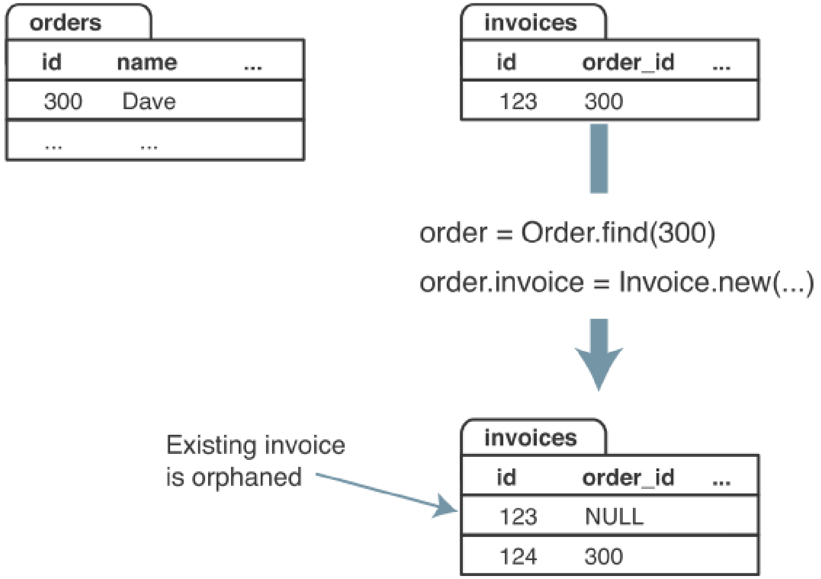
- :dependent=>:destroy – delete the associated item in other table when this is destroyed

- :dependent=>nullify – child row is orphaned

- :dependent=>false
  - child is not changed.

The :dependent option

```
class Order < ActiveRecord::Base
  has_one :invoice
end
```



orders

id	name	...
300	Dave	
...	...	

invoices

id	order_id	...
123	300	

order = Order.find(300)

order.invoice = Invoice.new(...)

invoices

id	order_id	...
123	NULL	
124	300	

Existing invoice is orphaned

Figure 19.1: Adding to a has_one relationship

A Real Estate DB Example – reflection on your lab

- 2 concepts
  - Home and owner
- Two tables
  - Homes
    - Street Number, Street, City. State. Zip
  - Owner
    - First, Last Name
- Foreign key
  - Owner has a home, or vice-versa

Two-table design

- Suppose we want to have multiple owners for a given house
  - Owner has home_id as a foreign key
  - Multiple owner records can refer to any given home

- Or, multiple homes for each owner
  - Home has owner_id as foreign key

- What if we want both?

Many-to-Many relationships: mapping tables

- A home can have many owners
- Owner can have many homes
- homes_owners – mapping tables
  - 2 columns: owner_id, home_id
- One entry for each instance of a home owner
- Can add more attributes
  - Purchase date?
  - Primary residence? – boolean
  - Order of ownership?
  - Ho do you do this? More on this later....

many-many relationships

- Home, owner related by has-many
  - homes_owners join table

```
def self.up
    create_table :homes do |t|
            t.column :street, :string
            ...
    end

    create_table :owners do |t|
        t.column :firstname, :string
        ..
    end

    …
```

the join table

```
create_table homes_owners, :id => false do |t|
    t.column :house_id, :integer
    t.column :owner_id, :integer
end

add_index :homes_owners , [:home_id, :owner_id]
add_index :homes_owners, :owner_id
```

- join table is named after two tables it joins (names alphabetized)
- join table does not need a primary key
  - compound key from two columns
- indices help speed queries

Indices

- Speed up database search
- Useful when we're dealing with something other than the primary key, and we're searching on it frequently
- In migration
  - add_index :table, :column
- For e.g., for the Orders model
  - add_index :orders , :name
- Generates SQL
  - Create INDEX name_index on orders (name);
- Composite indices – multiple columns – possible – pass column names as array
  - add_index :names, [:lastname, :firstname]

specifying many-to-many in models

- has_and_belongs_to_many

```
class Home < ActiveRecord::Base
   has_and_belongs_to_many :owners
...
end

class Owner < ActiveRecord::Base
   has_and_belongs_to_many :homes
end
```

- Note that there is no model for home_owners

has and belongs to many

- Owner has and belongs to many houses
- House has and belongs to many owners

Owner o = Owner.find(37)

houses = owner.houses

- Question – how do we handle attributes of this relationship?
  - purchase date
  - primary residence?
  - order – first, second, third, etc.

Attributes of many-many relationships

- Add columns to table
  - one column for each new attribute
  - many-to-many table is just like a regular table

- in Rails,
  - we treat join tables separately
  - have a table declaration in the DB
  - but no ActiveRecord...

- if we need attributes,
  - define a new model – with active record – and use it

many-many relationships with attributes

- Home and owner as before
  - define ownership table

```
create_table ownerships, :id => false do |t|
  t.column :home_id, :integer
  t.column :owner_id, :integer
  t.column :purchase_date, :date
  t.column :primary, :boolean
  …
end
```

- Add attributes to table…

Ownership ActiveRecord

```
class Ownership < ActiveRecord::Base
  belongs_to :home
  belongs_to :owner
end
```

- both Home and Owner has_many :ownerships

using many-many relationships with attributes

```
ownership = Ownership.new
ownership.purchase_date = Time.now
ownership.primary = true
ownership.owner = owner1
ownership.house = house3
ownership.save
```

- But, how can we get from owner to houses
  - and vice-versa?

has_many :through

- Used to indicate role of join classes that have active record implementations

```
class House < ActiveRecord::Base
    has_many :ownerships
    has_many :owners, :through => :ownerships
end

class Owner < ActiveRecord::Base
    has_many :ownerships
    has_many :homes, :through=> :ownerships
end
```

- now, we can say home1.owners << aNewOwner
- etc.
- The :through option can also have :conditions

Looking under the hood

- Efficient use of tools such as inferred relationships and accessors requires understanding of how the query generation works

- Put another way, this can lead to some woefully inefficient code if we're not careful

- Default in rails – defer loading child rows from the database until they are referenced
  - "lazy" loading

Example of deferred loading

```
class Course < ActiveRecord::Base
    belongs_to :professor
    has_many :students
end
```

Say you had to list all the course names, the professor teaching, and all the students

```
for course in Course.find(:all)
    puts “Course: #{course.name}”
    puts “professor: #{course.professor.name}”
    for student in course.students
            puts “student: #{student.name}”
    end
end
```

- 1 query for find
- Then if there are *n* courses, *n* more for professor
- *n* for students
- total of m students – m total for student name
- 2n+m+1 queries
- So sometimes you might want to override deferred loading

Alternative :include

```
for course in Course.find(:all, :include=> :professor)
    puts “Course: #{course.name}”
    puts “professor: #{professor.name}”
    for student in course.students
            puts “student: #{student.name}”
    end
end
```

- preloads professor
- pulls it in all at once
    - n+m+1 total

Alternative :include (cont.)

```
for course in Course.find(:all, :include=>[ :professor, :students])
   puts “Course: #{course.name}”
   puts “professor: #{professor.name}”
   for student in course.students
           puts “student: #{student.name}”
   end
end
```

- 1 query

Caveats with :include

- May or may not improve performance
- May slow things down – lots of data
- Use it only if you really need all of that data
- Try both with and without – benchmark
- Be aware of what is being done when you chase too many relationships

Look in the book/rails guides for finer points on

- when items get saved
  - save items manually and you won't hit problems
  - defensive programming

- caching counters
  - order.line_items.size will tell you how many
  - but there are ways to potentially speed this up
    - `:counter_cache => true` as a belongs_to option
    - `t.column :line_items_count, :integer, :default=>0` in the orders migration

Inheritance and ORM

- Object-Oriented world makes great use of inheritance

- Example: roles of people
  - person: name, email
  - customer: person with a balance
  - employee: person with a boss and a department
  - manager: employee with a group and a list of managed employees,
  - etc...

- How do we use relational databases to model these inherited relationships?
- First – how would you make the relational schema to represent this?

General idea: single-table inheritance (STI)

- all classes in hierarchy in one table
- column for each attribute of each class in hierarchy
- additional column - “type” - which type of object we're looking at

STI- migration

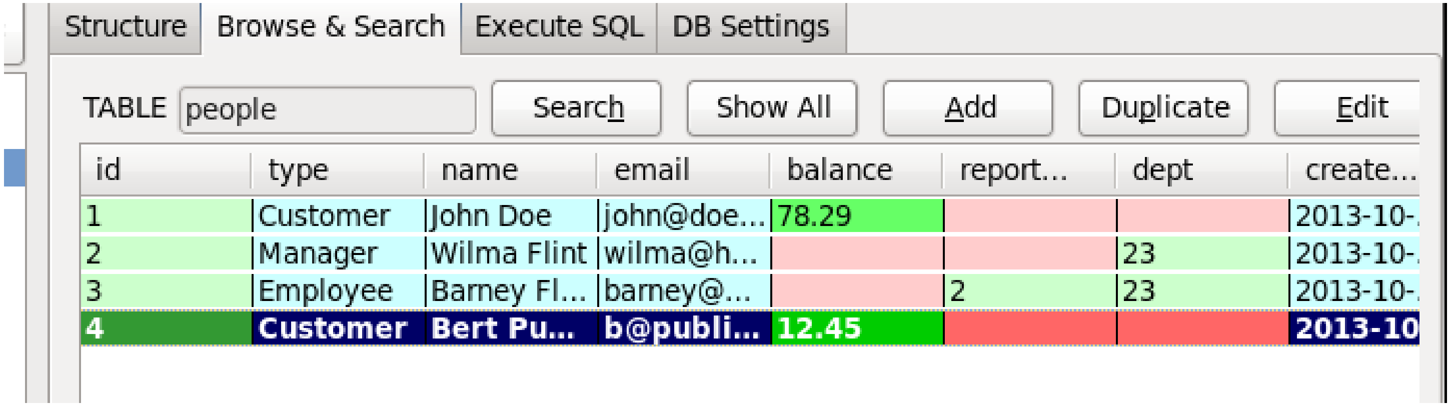
```
create_table :people, :force => true do |t|
    t.column :type, :string

    # common attributes
    t.column :name, :string
    t.column :email, :string

    # attributes for type=Customer
    t.column :balance, :decimal, :precision => 10, :scale => 2

    # attributes for type=Employee
    t.column :reports_to, :integer
    t.column :dept, :integer

    # attributes for type=Manager
    # - none -
end
```

Hierarchy of objects

```
class Person < ActiveRecord::Base
end

class Customer < Person
end

class Employee < Person
belongs_to :boss, :class_name => "Employee" , :foreign_key => :reports_to
end

class Manager < Employee
end
```

STI in use

```
Customer.create(:name => 'John Doe' , :email => "john@doe.com" , :balance => 78.29)
wilma = Manager.create(:name => 'Wilma Flint' , :email => "wilma@here.com" , :dept => 23)
Customer.create(:name => 'Bert Public' , :email => "b@public.net" , :balance => 12.45)
barney = Employee.new(:name => 'Barney Rub' , :email => "barney@here.com" , :dept => 23)


barney.boss = wilma
barney.save!
```



id	type	name	email	balance	report...	dept	create...
1	Customer	John Doe	john@doe...	78.29			2013-10-.
2	Manager	Wilma Flint	wilma@h...			23	2013-10-.
3	Employee	Barney Fl...	barney@...		2	23	2013-10-.
4	Customer	Bert Pu...	b@publi...	12.45			2013-10

STI in use

```
manager = Person.find_by_name("Wilma Flint" )
puts manager.class #=> Manager
puts manager.email #=> wilma@here.com
puts manager.dept #=> 23
```

ActiveRecord gives you the right type and sets it up correctly!

- Possibly wasteful in terms of space
  - probably no big deal if using text

Alternative to STI

- STI is good, but what if the different objects have common characteristics but share different representations?

- Polymorphic association
  - foreign key that links to objects of different types
  - along with an indicator of which type

- articles, sounds, and images all have a catalog_entry

Drops the table first

```
create_table :articles, :force => true do |t|
     t.column :content, :text
end

create_table :sounds, :force => true do |t|
     t.column :content, :binary
end

create_table :images, :force => true do |t|
     t.column :content, :binary
end
```

Polymorphic plumbing

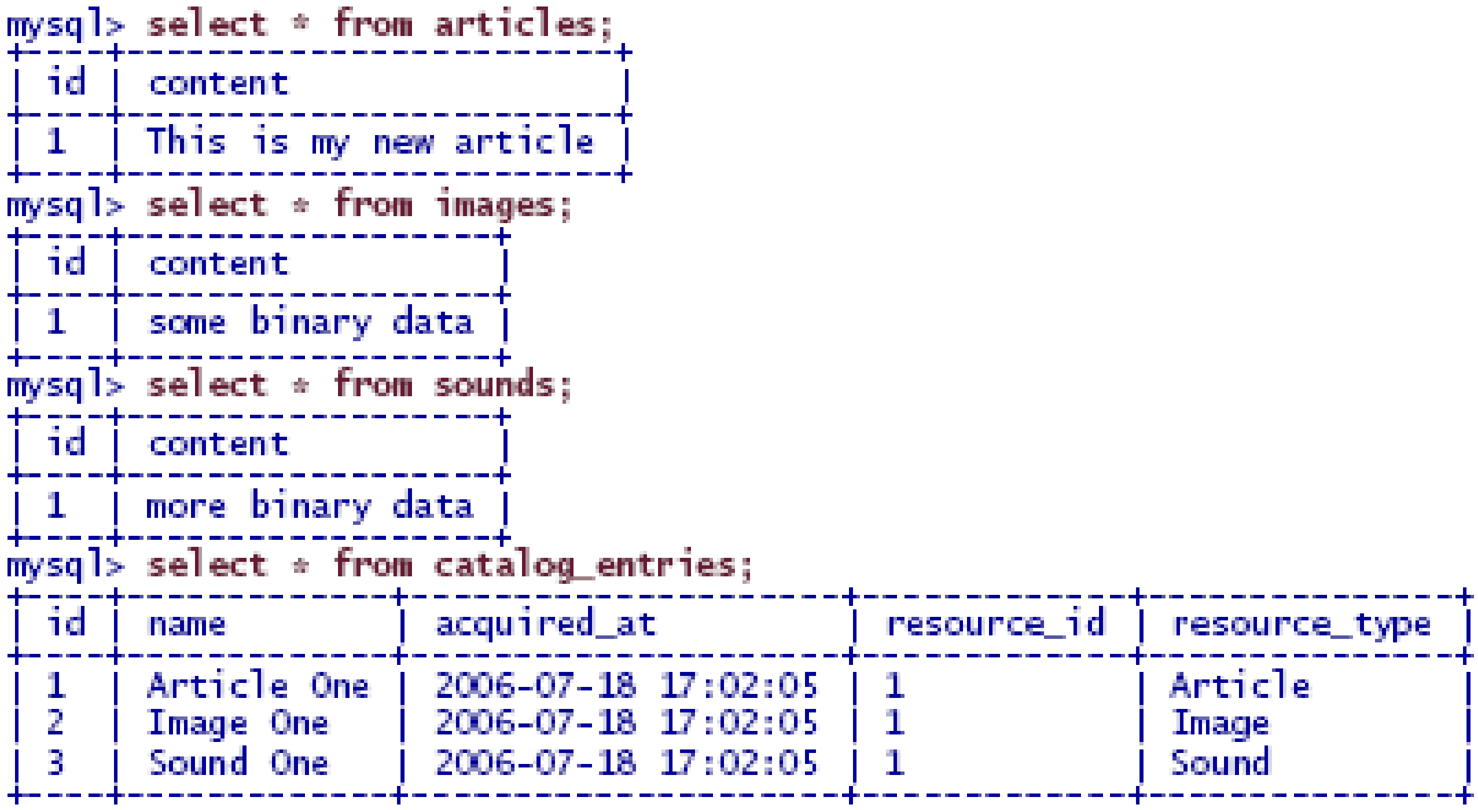
```
create_table :catalog_entries, :force => true do |t|
    t.column :name, :string
    t.column :acquired_at, :datetime
    t.column :resource_id, :integer
    t.column :resource_type, :string
end

class CatalogEntry < ActiveRecord: : Base
    belongs _to :resource, :polymorphic => true
end

class Article < ActiveRecord::Base
    has _one :catalog_entry, :as => :resource
end

ETC.
```

The combination of these is the foreign key

Polymorphic examples

```
c = CatalogEntry.new(:name => 'Article One' , :acquired_at =>
  Time.now)
c.resource = Article.new(:content => "This is my new article" )
c.save!

c = CatalogEntry.new(:name => 'Image One' , :acquired_at =>
  Time.now)
c.resource = Image.new(:content => "some binary data" )
c.save!

c = CatalogEntry.new(:name => 'Sound One' , :acquired_at =>
  Time.now)
c.resource = Sound.new(:content => "more binary data" )
c.save!
```

What do we get in the DB?



```
mysql> select * from articles;
+----+------------------------+
| id | content                |
+----+------------------------+
| 1  | This is my new article |
+----+------------------------+
mysql> select * from images;
+----+------------------+
| id | content          |
+----+------------------+
| 1  | some binary data |
+----+------------------+
mysql> select * from sounds;
+----+------------------+
| id | content          |
+----+------------------+
| 1  | more binary data |
+----+------------------+
mysql> select * from catalog_entries;
+----+-------------+---------------------+-------------+---------------+
| id | name        | acquired_at         | resource_id | resource_type |
+----+-------------+---------------------+-------------+---------------+
| 1  | Article One | 2006-07-18 17:02:05 | 1           | Article       |
| 2  | Image One   | 2006-07-18 17:02:05 | 1           | Image         |
| 3  | Sound One   | 2006-07-18 17:02:05 | 1           | Sound         |
+----+-------------+---------------------+-------------+---------------+
```

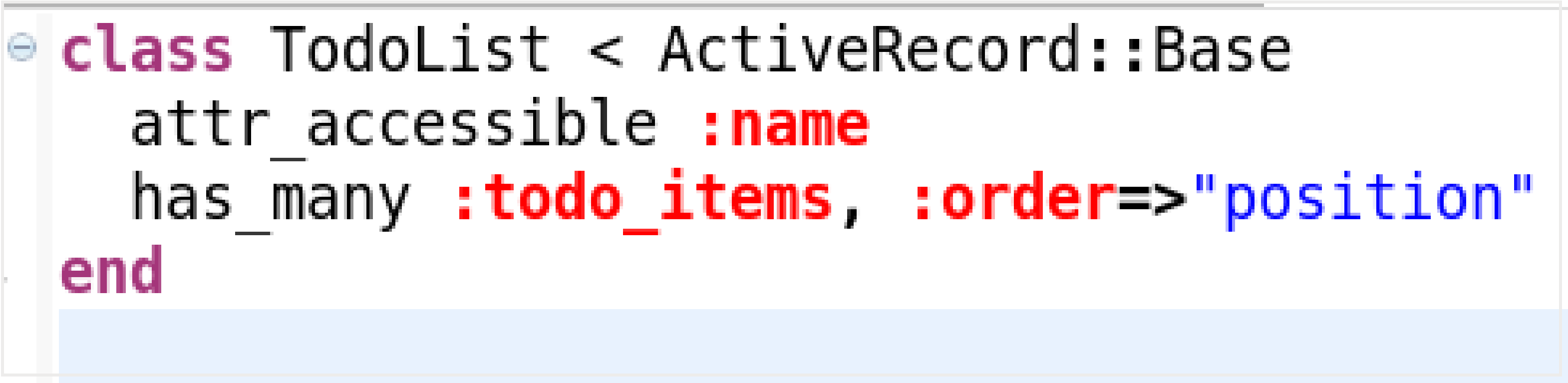
Self referential joins – show demo

```
class Employee < ActiveRecord::Base
    belongs_to :manager,
              :class_name => "Employee" ,
              :foreign_key => "manager_id"

    belongs_to :mentor,
              :class_name => "Employee" ,
              :foreign_key => "mentor_id"

    has_many :mentored_employees,
        :class_name => "Employee" ,
        :foreign_key => "mentor_id"

    has_many :managed_employees,
        :class_name => "Employee" ,
        :foreign_key => "manager_id"
end
```

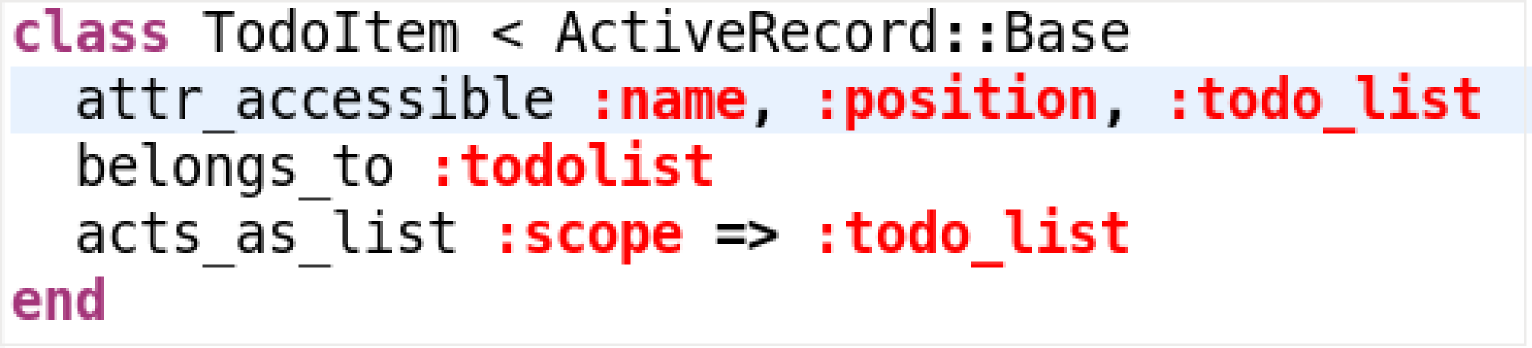
```
e1 = Employee.new (..)
e2 = Employee.create (..)
e3 = Employee.create (..)

e1.manager = e2
e1.mentor = e3
e1.save!

Queries:

e1.manager.name
e1.managed_employees.map {|e|
e.name}
       => [“dave”,”monica”]
```

Additional structures – *acts as List*

Install – gem ‘acts_as_list’ in gemfile

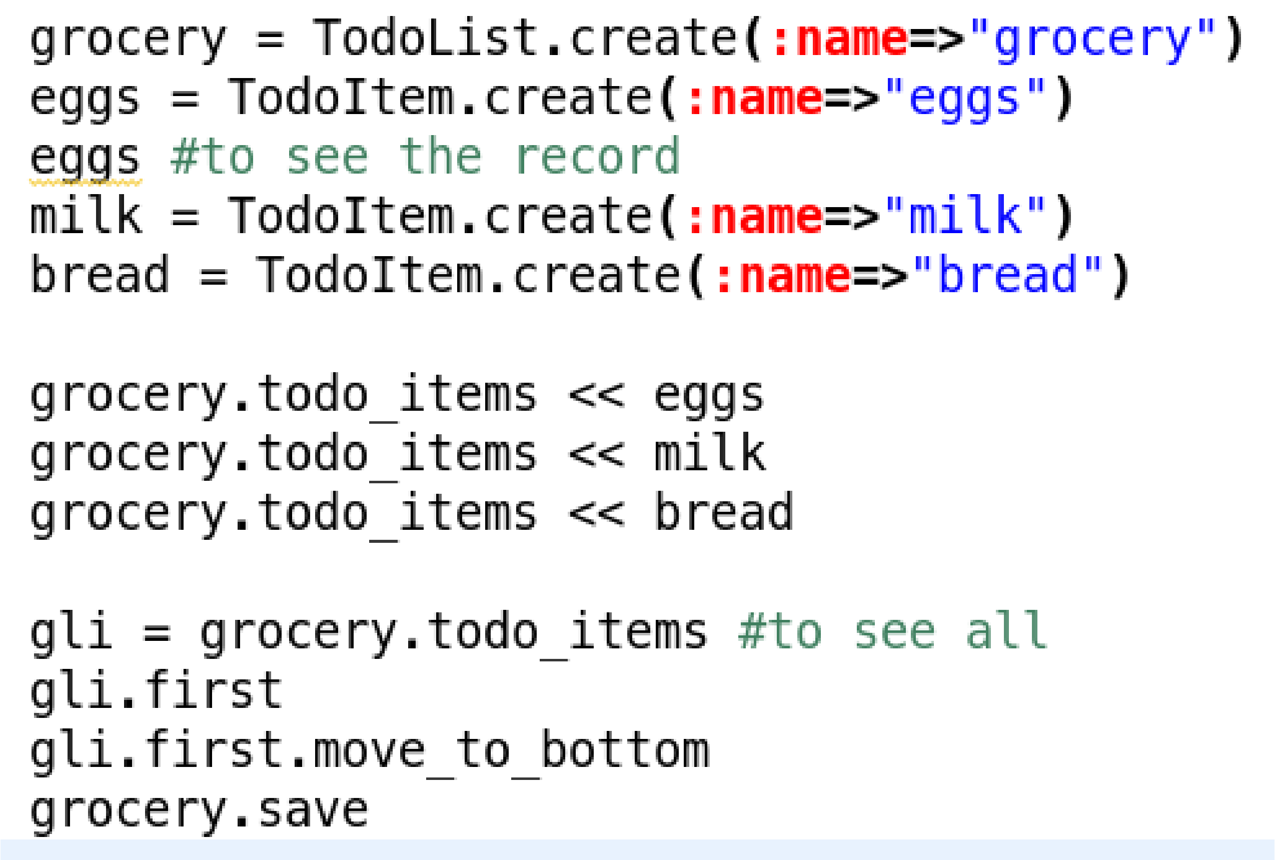


```ruby
class TodoList < ActiveRecord::Base
  attr_accessible :name
  has_many :todo_items, :order=>"position"
end
```



```ruby
class TodoItem < ActiveRecord::Base
  attr_accessible :name, :position, :todo_list
  belongs_to :todolist
  acts_as_list :scope => :todo_list
end
```

Acts_as_list

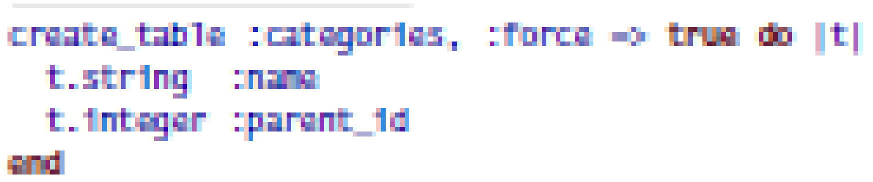


```
grocery = TodoList.create(:name=>"grocery")
eggs = TodoItem.create(:name=>"eggs")
eggs #to see the record
milk = TodoItem.create(:name=>"milk")
bread = TodoItem.create(:name=>"bread")

grocery.todo_items << eggs
grocery.todo_items << milk
grocery.todo_items << bread

gli = grocery.todo_items #to see all
gli.first
gli.first.move_to_bottom
grocery.save
```

Additional structures – *acts as Tree*

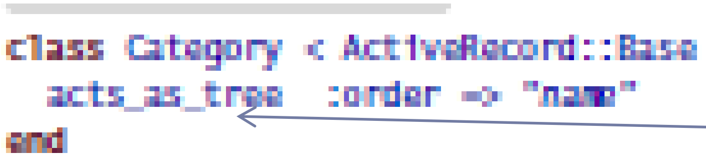
Sometimes – you need hierarchical relations in the DB

A good example is a category of books



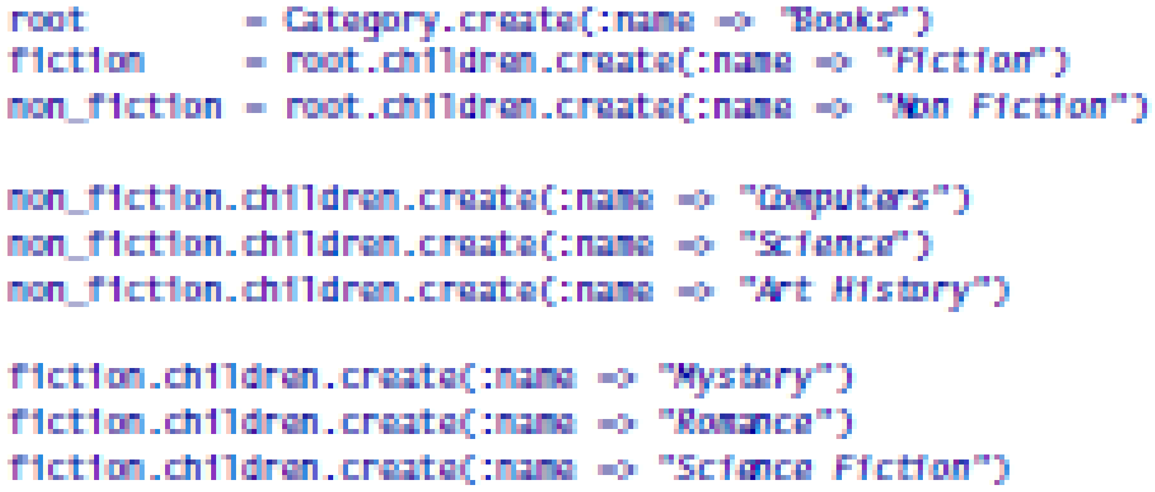
```
create_table :categories, :force => true do |t|
  t.string  :name
  t.integer :parent_id
end
```

Table - migration



```
class Category < ActiveRecord::Base
  acts_as_tree  :order => "name"
end
```

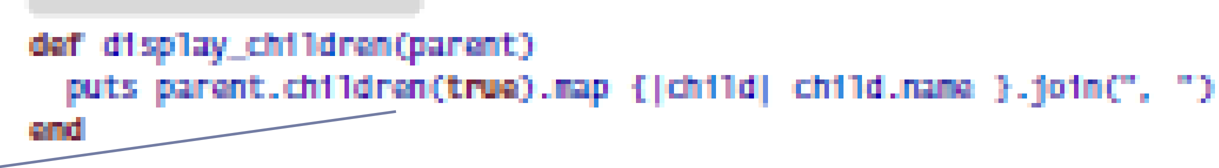
Model



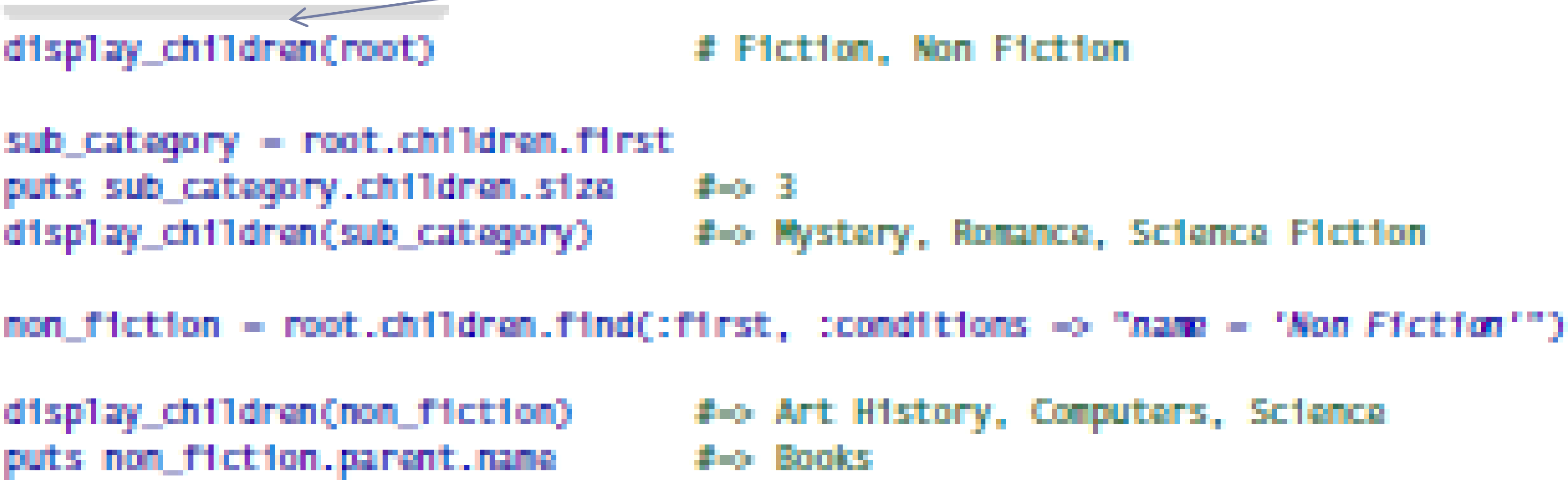
```
root        = Category.create(:name => "Books")
fiction     = root.children.create(:name => "Fiction")
non_fiction = root.children.create(:name => "Non Fiction")

non_fiction.children.create(:name => "Computers")
non_fiction.children.create(:name => "Science")
non_fiction.children.create(:name => "Art History")

fiction.children.create(:name => "Mystery")
fiction.children.create(:name => "Romance")
fiction.children.create(:name => "Science Fiction")
```

Populating

Additional Structures – *Acts as Tree*



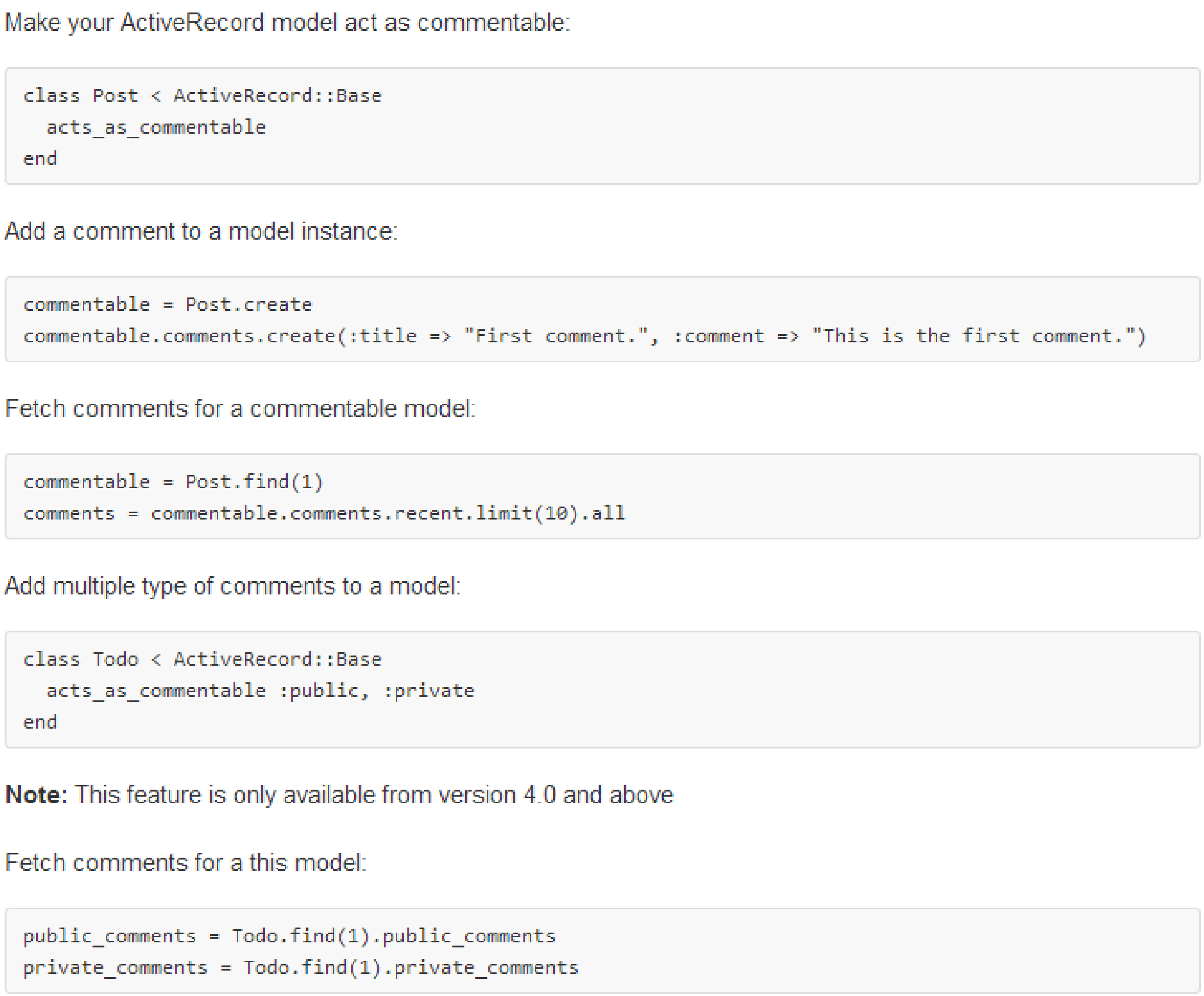
```ruby
def display_children(parent)
  puts parent.children(true).map {|child| child.name }.join(", ")
end
```



```ruby
display_children(root)              # Fiction, Non Fiction

sub_category = root.children.first
puts sub_category.children.size     #=> 3
display_children(sub_category)      #=> Mystery, Romance, Science Fiction

non_fiction = root.children.find(:first, :conditions => "name = 'Non Fiction'")

display_children(non_fiction)       #=> Art History, Computers, Science
puts non_fiction.parent.name        #=> Books
```

- Using the tree.
- See example and gem at https://github.com/amerine/acts_as_tree

acts_as_commentable (great for posts and comments)



Make your ActiveRecord model act as commentable:

```
class Post < ActiveRecord::Base
  acts_as_commentable
end
```

Add a comment to a model instance:

```
commentable = Post.create
commentable.comments.create(:title => "First comment.", :comment => "This is the first comment.")
```

Fetch comments for a commentable model:

```
commentable = Post.find(1)
comments = commentable.comments.recent.limit(10).all
```

Add multiple type of comments to a model:

```
class Todo < ActiveRecord::Base
  acts_as_commentable :public, :private
end
```

Note: This feature is only available from version 4.0 and above

Fetch comments for a this model:

```
public_comments = Todo.find(1).public_comments
private_comments = Todo.find(1).private_comments
```

- https://github.com/jackdempsey/acts_as_commentable

Generalizing

- Rails has interesting facilities for ORM
  - Hibernate, iBATIS, Entity Framework v2, ActiveObjects in Java etc. will be similar in flavor
    - if not in details
- Rails is easier to configure
  - defaults once again eliminate need for configuration files
- See readings for examples with Hibernate

Transactions

- Classic database problem
  - transfer $100 from account 1 to account2
  - account1.withdraw(100)
  - account2.deposit(100)

- must make sure that both happen or neither happens.

- Transactions – every SQL statement must succeed or all have no effect.

ACID

- atomic: all or none
- consistent: database is in legal state at beginning and at end
- isolated: no operation outside the transaction can see data in an intermediate state.
  - transaction history can be serialized.
- durable – once it's done it can't be undone.

Transactions at the DB level

- MySQL

```
start transaction
update account set balance = balance-100 where
  account_id=1
update account set balance = balance+100 where
account = 2
commit
```

- some bells and whistles – rollback forces undo
- Transactions are generally database-dependent

Transactions in rails

- Provided in Active Record

```
Account.transaction (account1, account2) do
   account1.withdraw(100)
   account2.deposit(100)
end
```

- Rails is smart about dependent/children items – makes them into a transaction

Race conditions

- Problem with multiple processes accessing a database
  - as is always the case with web apps
  - The last one to update *wins* the race
- p1 sets Order 456 name to "fred"
- p2 sets Order 456 name to "george"
- p1 saves
- p2 saves
- p1 changes are lost
- Not good.

Locking

- claim hold on a record until you're done with it

- Pessimistic locking can be an alternative to transactions
  - grab hold of everything you need
  - Do what you need to do
  - release locks

Optimistic Locking

- Before you save updated data
  - make sure row hasn't been changed
- Rails: Each row has a "version number"
  - checked before update – if they don't match, abandon & throw exception
- Enabled if table has a column called lock_version.
  - initialize to zero and leave it alone.

Optimistic locking example

```
create_table :counters, :force => true do |t|
    t.column :count, :integer
    t.column :lock_version, :integer, :default => 0
end

Counter.create(:count => 0)
count1 = Counter.find(:first)
count2 = Counter.find(:first)

count1.count += 3
count1.save

count2.count += 4
count2.save – this will throw an exception. thinking version 0, but count1.save made version 1
```

- You have to catch this exception in your code
- transactions rolled back before exception
- use this locking and transactions together

Transaction strategies

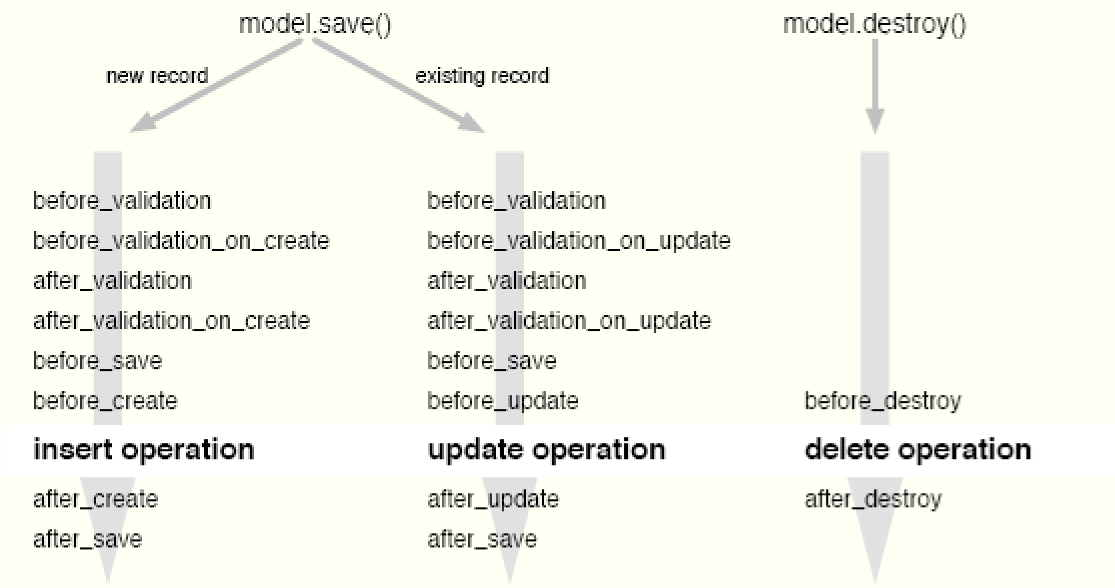
- Note – SQL transactions can include arbitrary tables

- Rails transactions only one class
  - Nest them to support more complicated logic
  - also useful for multi-database systems

- When to use transactions?
  - Interacting/dependent objects?
  - Sequence of changes that makes sense only as a whole

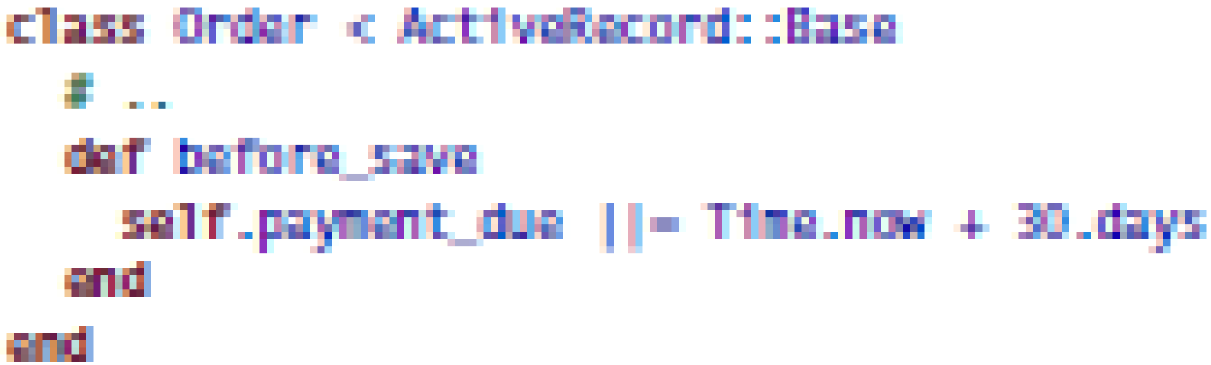
Callbacks

- Add code to appropriate points in the life cycle of an object
- 18 in pairs – before/after
  - after_find and after_initialize



Example with before_save

- A simple way to use a callback is to define a method itself.



```
class Order < ActiveRecord::Base
  # ...
  def before_save
    self.payment_due ||= Time.now + 30.days
  end
end
```

Defining callbacks

- appropriate procedure - before_save method for before save event
- Using handlers
- or, class to encapsulate callbacks for multiple objects
  - Say multiple objects need to normalize the credit card number before saving it to the database. Then define a common method and make all models use it.
    - Another example - encryption

Observers

- Transparently link to model class
  - register for callbacks without begin part of model

```
class OrderObserver < ActiveRecord::Observer
    def after_save(an_order)
            an_order.logger.info("Order #{an_order.id} created" )
    end
end
```

- XXXObserver associated with class XXX
  - Stored in app\models and call at runtime

- Install by specifying in application.rb

```
config.active_record.observers
    = :order_observer, :audit_observer
```

Comments on Callbacks and Observers

- Use callbacks to encapsulate common processes in object life cycle
  - encrypt before save
  - decrypt after read
- observer
  - logging and other subsidiary tasks