

Notes for Exam 1 in 67-272

Data Modeling

A **bridge** to convert requirements into database

Can be done **early** in the process

Cheaper to fix errors at this stage

Understandable to user and developers

Data is **critical!**

Entity-Relationship modeling is fairly easy to do

Entity

The **Nouns** (possibly and attribute)

Entities have **attributes**

Entities have **relationships** with other Entities

Relationship

The Verbs

"Employees work at Shop"

shop has_many :employees

employee has_many :shifts

employee has_many :shops, :through => :shifts

Type:

One-to-One (hard line)

One-to-Many (hard line & crow's feet)

Many-to-Many (crow's feet)

Optional (open circle)

Required (hard line)

Associative Entity

Entity which relates two other entities

(can create many-to-many relationships)

Databases cannot handle many-to-many . . . very well

ERD

When creating an ERD, identify all entities and attributes

Define relationships between entities

Determine connectivity and transform many-to-many relationships

Ascertain whether required/optional

Recognize that data modeling is usually iterative process

Need to go back and make changes several times
This is much cheaper than going back later in development

Normalization

Flat File (Excel Spreadsheet) vs. **Relational Database** (RDBM)

Flat File: hard to process, repetition, error when changing, loss of IQ

Relational Database: Easy to process, update, understand

Database

Table (file) Entity

Record (row) Instance of Entity

Field (column) Attribute

Keys:

Primary: Uniquely identify a record in a table

(id auto_increment not null, (primary key id));

Foreign: A field in Table A which is also a primary key in Table B; used to establish links between tables.

Composite: A combination of keys which together serve to uniquely identify a record. (nowadays you can jump off a cliff before worrying about saving space by combining keys... just create another id)

Florida keys: Small chain of islands off the coast of Florida

When referring to database tables, pluralize (you are referring to many instances of the table entity)

Primary Key

Composite Key (just don't use these)

Foreign Key

Database Integrity

1st type: Entity Integrity

Key Idea – table must have a valid primary key

2nd type: Domain Integrity

Key Idea – data type and format must be valid

Use Validations before update, insert, create, save etc

3rd type: Referential Integrity

Key Idea – don't leave behind orphaned records

Normalization

Process to create a flexible, nonredundant, and efficient data model that can be implemented in a RDB

Helps preserve referential integrity

Usually problematic if < 3NF (Third Normal Form)

Sometimes ok for "sensible denormalization"

5 degrees of Normal Form

Can go higher than 3NF but 3NF is ok

3.5 Questions to get to 3NF

-Does the entity have any repeating elements?

Have two entries for one attribute

(1st Normal Form)

-Does the entity have a composite key?

If no, go on to 3.

Else, Are there any partial dependencies

Is there an attribute which only depends on ONE of the composite keys? If so, pull those attributes out and make own table.

-Are there any transitive dependencies?

Are there any fields in the table that depend on another field in the table that is not the primary key?

Kinda like Q2.a. Pull those attributes out

Denormalization

Address, phone data

City, State, Zip Code

Why Denormalize?

Easy queries

Less processing

Database Normalization checks your ERD

If you have a good ERD, you probably don't need to go through normalization

***Always **note** deviations from Normal Form in your Data Dictionary & Data Model

SQL

"Structured Query Language (SQL) "sequel" is a language that provides an **interface to relational database systems**. It was developed by IBM in the 1970s for use in System R. SQL is a de facto standard, as well as an ISO and ANSI standard."

ISO & ANSI => organizations to keep standards...makes our lives easier

SQL Allows for:

- Data extraction (SELECT)
- Data Manipulation (INSERT, UPDATE, DELETE)
- Data Definition (CREATE, DROP, TRUNCATE)
- Data Control (GRANT, REVOKE)

Queries:

Order Written	Order Processed
SELECT	FROM
FROM	WHERE
WHERE	GROUP BY
GROUP BY	HAVING
HAVING (Can only use if 'group by')	SELECT
ORDER BY	ORDER BY
LIMIT (Auto 30)	LIMIT

- From: Get all these tables and JOIN them together
Take first row and join with every other row with other table
Joins will create a really big table
- Where: Pick out the rows which meet Where criteria
- Group By: Reorder the data or Sort rows by criteria
- Having: Now check having criteria
- Select: Now, only get the fields I originally asked for
- Order By: Sort
- Limit: Apply high level complex iterative algorithm which shuffles the data.

RoR

Ruby on Rails or Rails is a development framework created using the Ruby programming Language by David Heinemeier Hansson

Goods:

- Quick Development
- Easy Development (A lot is already done for you)
- Available Plug-ins
- Organized
- Best Practices
- Consistent Methodologies (easy for multiple people to work on over time)

- Well Documented
- Application Architecture (Model View Controller :: MVC)

Bads:

Learning Curve

Strict Rules & Methodologies (Spelling, order, pluralization. All Matter)

Also part of Goods when you master these techniques

High Overhead (Rails has a lot of code on the back end to make everything work)

Ruby

How to **pick up New Languages** (cause they keep changing)

Develop plan and identify recourse for learning new language

Handling basic data types and structures in new language

Using control structures (loops, conditionals)

(just to get a good feel for the language)

Creating classes and objects

How to work with OO programming in new language

Utilizing data storage

Include files and libraries

Don't reinvent the wheel

Debugging and handling exceptions

Using regular expressions

Philosophy of Ruby

"For me, the purpose of life is, at least partly, to have joy. Programmers often feel joy when they can concentrate on the creative side of programming, so Ruby is designed to make programmers happy."

-mats (creator of Ruby :: Yukihiro Matsumoto)

Three Principles

1. **Conciseness** – Writing code in Ruby should involve the minimum amount of commands necessary. Code should be terse but also understandable.
2. **Consistency** – Ruby coding should follow common conventions that make coding intuitive and unambiguous
3. **Flexibility** – There is no one right way. You should be able to pick the best approach for your needs and be able to even modify the base classes if necessary.

These three together lead to an important concept in Ruby – **the principle of least surprise**

Everything in Ruby is an Object: Even numbers

That means they can have methods

```
500.times {puts "yo"}
```

But you can do this many other ways:

```
(1..500).each { |i| puts "yo"}
```

Or

```
For i in (1..500) do
  Puts "#{i}. yo"
End
```

Ruby Basic data types:

Strings, Numbers, Arrays, Hashes, Buffers (stacks & queues), M-way trees, Heaps

Model View Controller

Architecting Software

Needs to be:

Understandable

Extensible

Many different architecture patterns exist

(Feel ambitious? Read Patterns of Enterprise Application Architecture by Martin Fowler)

MVC is a System Architecture

MVC is **not just for Web Development**

MVC can be applied to **any Programming Language** and **any Development Platform**

MVC in Rails

What do we do?

First, we have a request (from browser :: aka url is entered)

Ok, so what do we do with the URL

Look at the URL... but who?

Rails "Routing" looks at URI and processes it

The Rails "dispatcher" (in routing) says:

"I know exactly what to do with this because...."

URL = "http://www.mysite.com/object/1"

STOP. That's not all. HTTP also comes along with something called a "HTTP Verb" or (Request Method)

This means that I have something in addition to the URL

The HTTP Verbs/Request Methods we use in Rails are:

Get
Post
Put
Delete

*This is part of REST (also follows CRUD)

This means that we can have the exact same URL but they are still different (very different :: one can show a record while the other deletes it)

So, moving along...

Get rid of the "http www .com" crap

Now we have: "/object/1" and the HTTP Verb "GET"

*Note: Even though the browser doesn't display the HTTP Verb in the URL, It is still being sent to the server in the request

So our dispatcher knows to go to the ObjectController

But the Rails dispatcher looks at the HTTP Verb and maps it to the correct **:action**

HTTP Verb	Action
GET	Show
DELETE	Destroy
PUT	Update
POST	create

So now the dispatcher can tell the ObjectController to invoke the "show" :action

Oh, and by the way, here is some ID = '1' in case you need it (cause the dispatcher doesn't know what the ID is :: the controller will handle it from here, it just needs to get all the info first including that ID)

Plurals vs. Singular in Rails

And Object or Model is SINGULAR

However, when we refer to the table for that Model, we Pluralize

Task *model*
tasks *table*
TaskController

TaskController interacts with Task (model) which uses the tasks (table)

Automated Tasks in Rails

rake db:create
rake db:migrate (rake db:migrate VERSION=0)
ruby script/generate (generate what scaffold, model, controller)
ruby script/server
ruby script/plugin

Model basics in Rails

Models rely on ActiveRecord

ActiveRecord is the Rails ORM

ORM?

Object-Relational Mapping: This is how we make rails database independent. The ActiveRecord ORM keeps track of the layer between the Model and the Actual Database. (therefore, developers do not have to worry about any database management or SQL)

Models (by definition in the MVC architecture) contains ALL the business logic

This means you should never be checking or validating or anything that is defined as "business logic" within the Controller (and certainly not the view)

This means, Rails Models have:

Basic Validations
Other Methods

Models hold all the data & business logic

Rails Controllers

Controllers are in charge

Basic actions include: index, show, new, create, edit, update, delete

Rails creates these actions for you but this is not DRY

DRY (Don't repeat yourself)

There are parts of new and edit which are the same

Rails Assumptions

Rails assumes that there is a template for each "action"

For the show :action there is a show.html.erb template
(you can change this)

Controllers handle the Process Logic

Rails Views

CSS and images
Layouts
Partials

Controllers puts all these parts together based on the instructions from the dispatcher (the requested URL)

Views can only use the data which the Controller gives them
Can't try adding/changing the use of data in the view without appropriately changing the controller.

Rails

```
CREATE TABLE Camps (  
  id INT UNSIGNED AUTO_INCREMENT NOT NULL,  
  start_date DATE,  
  end_date DATE,  
  session INT,  
  cost INT,  
  curriculum_id INT,  
  active INT,  
  Primary Key (id)  
);
```

```
C:\Ruby> ruby script/generate model camp start_date:date end_date:date  
session:integer cost:integer curriculum_id:integer active:boolean
```

	db2	mysql	openbase	oracle
:binary	blob(32768)	blob	object	blob
:boolean	decimal(1)	tinyint(1)	boolean	number(1)
:date	date	date	date	date
:datetime	timestamp	datetime	datetime	date
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float	number
:integer	int	int(11)	integer	number(38)
:string	varchar(255)	varchar(255)	char(4096)	varchar2(255)
:text	clob(32768)	text	text	clob
:time	time	time	time	date
:timestamp	timestamp	datetime	timestamp	date

	postgresql	sqlite	sqlserver	sybase
:binary	bytea	blob	image	image
:boolean	boolean	boolean	bit	bit
:date	date	date	datetime	datetime
:datetime	timestamp	datetime	datetime	datetime
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float(8)	float(8)
:integer	integer	integer	int	int
:string	(note 1)	varchar(255)	varchar(255)	varchar(255)
:text	text	text	text	text
:time	time	datetime	datetime	time
:timestamp	timestamp	datetime	datetime	timestamp

Note 1: character varying(256)

Migrations

When, and I repeat when, we screw something up in setting up models, we are going to want to know a little bit about migrations

Migrations = screwing with database (always fun times)

To appreciate Migrations you should try developing an application from scratch and then making changes and then having something go wrong and then being pressured to get everything working again and then jumping off a cliff...

Or just understand that rails migrations (along with all these other confusing things in the rails framework) are really helpful in the "real development world"

Hokay, so here's the Earth:

```
Ruby script/generate model my_freakin_model name:string
```

Will create a Migration...

Rails will also put a time stamp on this migration

So we can roll back changes made to database in a semi-structured manner.

Hypothetical situation: you just accidently ran a script/generate (you wanted to just create model and typed scaffold

Or you put in the wrong attributes and wrong types)

You can roll back to the desired version (desired timestamp)

**

```
rake db:migrate VERSION=20080906120000 (the timestamp)
```

or

```
rake db:migrate VERSION=0 (to go back to square one)
```

the migrations which are created by rails during a ruby script/gen... call can also be created *manually* by just creating a new migration file with the appropriate settings (NetBeans can help you with this)

then you need to define what the migration does:

(ruby script/generate model Product name:string description:text)

(if you wanted to do it through terminal)

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end

  def self.down
    drop_table :products
  end
end
```

Up vs Down

Hokay, so... when you *rake db:migrate*

 Rails runs all the "up"'s for all migrations

However, you you roll back (*rake db:migrate VERSION=82374934739*)

 Rails runs the "down"s for all the migrations with a timestamp
after the VERSION timestamp)

So you define the create process in "up"

And you do the just-in-case-i-need-to-go-back in the "down"

Making changes to Model

(AKA I need to add an attribute to the Model)

(AKA I need to add a column to the table)

(AKA I really like the way rails knows I'm gonna screw up)

You can just create another migration like the one below:

-add a new attribute "part number" to the Product model

```
ruby script/generate migration AddPartNumberToProducts
```

-which translate to:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

*Ok so stop and think about this

We just created a blank migration (that wasn't that helpful)

But if we name the migration file correctly (AddXXXToYYY) and then add the appropriate attributes to the end you can do a little more:

```
ruby script/generate migration AddPartNumberToProducts part_number:string
```

```
-we will create:
class AddPartNumberToProducts < ActiveRecord::Migration
  def self.up
    add_column :products, :part_number, :string
  end

  def self.down
    remove_column :products, :part_number
  end
end
```

We can also do a similar process with *r s/g m RemoveXXXFromYYY name:type*

<http://guides.rubyonrails.org/migrations.html>

More on the Model

Normal Definition:

```
Class Product < ActiveRecord::base
End
```

Validations:

- make sure only valid data is recorded

 - So things will work as intended in all areas of application

But you can make validations in many places

- cannot put these operations directly in the database since this will make the application database dependent and hard to test and maintain (especially in database changes)

- cannot do this client side (AKA that Quick AJAX response using Javascript) because Javascript may be disabled in some browsers or other problems may occur.

- therefore, it may be a pretty good idea to do these validation operations in the ActiveRecord Class ... aka the Model.

We have some built in validations which only need a few parameters

```

Class Product < ActiveRecord::Base
  validates_presence_of :title, :description, :image_url
  validates_numericality_of :price
  validates_uniqueness_of :title
  validates_format_of :image_url, :with => %r{\.(gif|jpg|png)$}i,
  :message => 'must be a URI for GIF, JPG' +
  'or PNG image.(gif|jpg|png)'
End

```

As you can see above, these built in validations can easily be used to validate one attribute or many attributes (separated by commas)

You can also define your own validation:

```

protected
def price_must_be_at_least_a_cent
  errors.add(:price, 'should be at least 0.01') if price.nil? || price < 0.01
end

```

Note* a few things:

“protected” means that ANY METHOD BELOW the “protected” declaration can only be used within the context of this Model (... only instances of this Object can use it)

“errors” has a method “add” which needs to know which attribute has the error “:price” and the message which should be displayed

And then the conditions on which this should happen if nil? Or less than .01

Now* we can call this custom validation with our other built-in validations

```

class Product < ActiveRecord::Base
  validate :price_must_be_at_least_a_cent
end

```

**Note the syntax here

This is only a brief introduction to ActiveRecord Validations. For more >>

http://guides.rubyonrails.org/activerecord_validations_callbacks.html

Callbacks (similar to validations)

-Callbacks are like validations but can be called at different stages in the Object's lifecycle.

```

class User < ActiveRecord::Base
  validates_presence_of :login, :email

  before_validation :ensure_login_has_a_value

```

```

protected
def ensure_login_has_a_value
  if self.login.nil?
    self.login = email unless email.blank?
  end
end
end
end

```

*note the syntax of: "before_validation :ensure_login_has_a_vlaue"

Can also use:

```

before_create :something_here
before_filter :authorize, except => :login
before_destroy

```

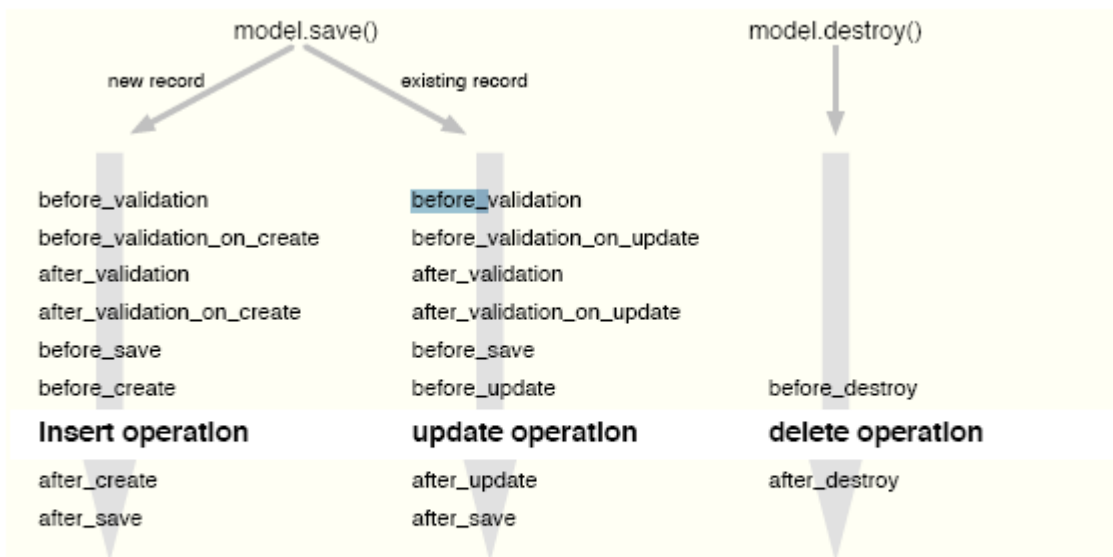


Figure 19.1: Sequence of Active Record Callbacks

Conditional

You can make validations & callbacks conditional by adding:

```
, :if => some_method?
```

Aka

```

class Order < ActiveRecord::Base
  before_save :normalize_card_number, :if => :paid_with_card?
end

```

Named_scope

-Probably one of the more kinda-cool-things-about-rails
Why? It's like having a whole box of bubble wrap

Let's say I wanted to add methods in the a Model to execute some query:

```
class Region < ActiveRecord::Base
  #other code
  def self.ordered
    find(:all, :order => "name ASC")
  end
end
```

We have to define a whole new method for this
And if we wanted to make more complex queries?????
More methods!
**So How do Named_scopes help?

First, we can just declare a named-scope to do the same as above:

```
class Region < ActiveRecord::Base
  named_scope :ordered, :order => "name ASC"
end
```

Ok great, we rewrote a method in less lines AND we can just like any other method

But what if we wanted to add more constraints?

Well, we define named_scopes in sub parts (DO NOT define a complex named_scope)

Because you can "chain" named_scopes together to create more complex queries

Which means you can combine them in the appropriate order
(if you understand how totally awesome this is..... you are a NERD)

Let's take a look:

```
class Region < ActiveRecord::Base
  belongs_to :country
  named_scope :ordered, :order => "name ASC"
  named_scope :by_country, lambda { |c| { :conditions => ["country_id = ?", c.id] } }
}
```

```

    named_scope :containing, lambda { |s| { :conditions => ["name like ?",
"%#{s}%" ] } }
end

class Runner
  c = Country.find_by_name("Canada")
  regions = Region.by_country(c).containing("A").ordered
  puts "Provinces in Canada containing the letter A in ascending order:
#{regions.inspect}"
end

```

in the Runner class, we chain named_scopes (outside of the mother class) to create the following query:

```
SELECT * FROM `regions` WHERE ((name like '%A%') AND (country_id = 1)) ORDER BY name ASC
(but you never see that as a rails developer)
```

*Also, named_scopes come in handy when creating drop downs

THINK about creating a new Chess Camp and putting a drop down selection box in the form to create a new camp for curriculums (no really think about this, you will need to do it)

You can use a named_scope to get the appropriate curriculum names and id's (as in get the "active" curriculums...)

<http://mattpayne.ca/blog/post/named-scope>

Active Record wtf?

Once again, you will need to develop applications from scratch to truly appropriate things like Active Record.

ActiveRecord is Rails' way of linking Models to the actually Database

It is that extra layer which abstracts some things out of your way.

ActiveRecord knows to link "integer" to int in some DBs and number or integer is other DBs

Active Record knows to link "Boolean" to tinyint or some other integer/Boolean representation in DBs

ActiveRecord lets you create relationships between Models (tables/objects) So how does ActiveRecord relate to Model?

A Model Class is a sub class of ActiveRecord

Hence the "< ActiveRecord::Base" in each model class

Since a Model is part of ActiveRecord and ActiveRecord keeps track of relationships/associations..

We define model relationships in the model.. with the whole:

```
Has_many
Belongs_to
Has_one
Has_and_belongs_to_many
Has_many :things, :through => :other_thing
```

So wtf does this do?

“These declarations enable a good bit of automatic behavior. For example, if you have an instance variable `@post` containing a post, you can retrieve all the comments belonging to that post as the array `@post.comments`.”

http://guides.rubyonrails.org/getting_started_with_rails.html#_associating_models

along with Relationships:

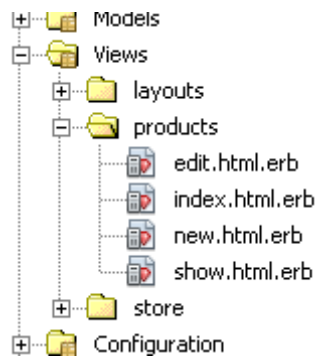
Routes are entries in the `config/routes.rb` file that tell Rails how to match incoming HTTP requests to controller actions. Open up that file and find the existing line referring to `posts` (it will be right at the top of the file). Then edit it as follows:

```
map.resources :posts, :has_many => :comments
```

***This creates `comments` as a *nested resource* within `posts`. This is another part of capturing the hierarchical relationship that exists between `posts` and `comments`.

The view

The view is pretty straight forward . . . until you want to DRY up your code and include things like partials and stuff like that.



And then organize all this crap (different pages for different actions for different controllers etc.)

```
appname\app\views\*
  layouts\*
    products.html.erb
    etc.
  products\*
    edit.html.erb
    etc
```

**Every Controller is assumed to have a view associated with it
(in the case above there is probably a ProductController which will use the views\products\
*.html.erb files for the actions defined in the ProductController)**

**So It is clear why we create a new folder under “views” for each controller
And then create a new file for each action (which needs a display of sort)**

Well we also have “layouts” under “views”

*note that the *.html.erb files do not contain your needed html tags such as
<html> or <body>

That stuff is kept in “layouts”

So let’s trace the App so far:

Let’s call the file under /views/layouts/product.html.erb >> **“model template”**

Let’s call the file under /views/products/show.html.erb >> **“model action piece”**

Browser_request>dispatcher>controller

Controller pulls data from Model

Controller gets the “model action piece” and gives it the data

Controller then send “model action piece” to the “model template”

Controller then sends them all back to browser

Template because it creates the general structure and look & feel of the page

It also has the “<%= yield %>” tag in it

(which is where the model action piece will go)

Action Piece because it is *probably* tied to an :action in the controller

It holds all the instructions to display the data give (by controller from
model)

And it is a “piece” of the whole page

The piece that fits into the “yield” tag

The **Model Template** (*aka* app/views/layouts/products.html.erb)

May want to include stylesheets:

<%= stylesheet_link_tag 'scaffold', 'depot' %>

May also want to determine where to put any messages;

<%= flash[:notice] %>

And then determine where to put the main content with:

<%= yield %>

The **Model Action Piece** (*aka* app/views/products/new.html.erb)

Show all the data (*aka* iterating over the collection)

```
<% for product in @products %>
```

```
<% end %>
```

Display Image

```
<%= image_tag product.image_url, :class => 'list-image' %>
```

Alternate Stylesheet class definition (*aka* alternate the color of rows)

```
<%= image_tag product.image_url, :class => 'list-image' %>
```

Use the "product" variable (represents the current product instance in the iteration) to display attribute values:

```
<%=h product.title %>
```

(*the **hO** to remove any special html tags which will cause problems)

Link to something

```
<%= link_to 'New product', new_product_path %>
```

Link to an action associated with instance of object:

```
<%= link_to 'Edit', edit_product_path(product) %>
```

Or

```
<%= link_to 'Destroy', product,
```

```
:confirm => 'Are you sure?',
```

```
:method => :delete %>
```

Day 1

Tuesday, January 13, 2009

1:33 PM

Get Books

Got to website and of PDF of link

Figure out Lab times

Project 1

Phase 1

http://rook.hss.cmu.edu/~67272/files/chess_camp_narrative.pdf

Read for phase one

Find Entities: Nouns **

Next Day

Thursday, January 22, 2009
10:32 AM

Select DISTINCT * FROM

NOT IN

```
select en.class_id, count(*)  
from enrollments as en  
group by en.class_id  
Having count(*) >= 2
```

```
select concat(stu_lastname, " ", stu_firstname) as name,  
case when credits is null then 'N/A'  
else credits end as Credits  
from students  
order by stu_lastname;
```

```
CASE WHEN *** THEN ***  
ELSE *** END AS ****
```

```
select cl.id as Course, Count(en.class_id) as enrollment from classes as cl  
left join enrollments as en  
on cl.id = en.class_id  
group by en.class_id  
order by cl.id;
```

```
select round(avg(st.credits),2) as "avg mat stud credits" from students as st  
where major_id = (select id from majors where major_name = 'Math');
```

Other day

Tuesday, January 27, 2009
10:29 AM

Left vs Outer Join:

All left joins are Outer Joins
Some packages make you say "Left Outer Join"

Also have Full Join

Which gives you everything even without matches

If primary and foreign key are the same

And your database package implements "using"
You can use the field names with "using"

Cross Join...

Self Join:

Join table to itself
Give table two different alias
And join on self
....

Union:

Get from multiple tables
Example: get all users' first & last name

Can also use Union All to have redundancy

****Field names have to be the same****
Columns must be the same

"union" will eliminate duplicates
****only if all columns match :: first_name & last_name etc**

Intersect or Join

Only the matches ... in both

Except :: all queries in first but not in second

Another Day

Thursday, January 29, 2009
10:38 AM

Architecting Software:

- Understandable
- Extensible

Model View Controller:

Browsers sends request >> request goes to Controller (**Dispatcher** takes request and **Routes** to correct **controller** with correct **Action**)

Controller takes action and **processes** by accessing **Model**

Model returns data back to controller

Controller takes **data**, **processes**, then **sends** to **view**

View displays to browser

Controller Accesses the Model (Model gets data)

Controller then gives data/information to the view which displays back to the browser

OREO:

Model: all about taking care of business

Model: knowing what to do

Carry the business logic

Knows how to eat the oreo

Views: you can dress up your oreo :: many different ways

However, there are some common features to oreo's / views

OREO is on every cookie

Reusable parts == "Partials"

Models know everything :: Views know nothing

Controllers put it all together :: Cream filling (variation in controller/filling)

We want controllers to be skinny

Controllers have Logic >> Process Logic

NOT BUSINESS LOGIC

Controller is like the **Traffic Cop**

MVC Benefits

Distribute Work Load

Put people where their strengths are

Back end / front end / and cream

Work in Parallel

Testing:

Can test separate parts

Interchange Code

Change Database and only change model

Can actually extract model from database

Create stored procedures so you can have database guys and model guys

SQLite (Development) to MySQL (Production)

Then switch to Oracle

Can easily (more easily) be done

Rails is a opinionated software

Things work in a certain way within a Framework

Framework has a **specific layout** and **methodologies**

Framework relies on **Structure**

The following day

Thursday, January 29, 2009
10:38 AM

SQL

Sunday, February 08, 2009
9:28 PM

Order written:

- SELECT
- FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- LIMIT

Order processed:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- ORDER BY
- LIMIT

Exam 1

Thursday, February 12, 2009
10:39 AM

Interpret SQL

USE Database from Phase 1.b

Write sql

Using sub queries

Join

Some you cannot use where

Rails

Matching

Normalization

3-5 questions

Rails: gives a named_scope and must generate the SQL

Active Record ORM/OMR

Go look at script server as it prints out the SQL that is being generated from within the rails application

Some terms:

Before Validate :callback

After :callback

Callback

>>Models

#Call back

Relationships

#named_scopes

#

Controller Equivalent

-Filters

-Before

-After

-Around

Before_filter :login_required :except => "index"

Or

Before_filter :login_required :only => "page"