README.rdoc

README.rdoc
Last Update: 2011-09-16 22:39:19 -0700

Sequel: The Database Toolkit for Ruby

Sequel is a simple, flexible, and powerful SQL database access toolkit for Ruby.

- Sequel provides thread safety, connection pooling and a concise DSL for constructing SQL queries and table schemas.
- Sequel includes a comprehensive ORM layer for mapping records to Ruby objects and handling associated records.
- Sequel supports advanced database features such as prepared statements, bound variables, stored procedures, savepoints, two-phase commit, transaction isolation, master/slave configurations, and database sharding.
- Sequel currently has adapters for ADO, Amalgalite, DataObjects, DB2, DBI, Firebird, IBM_DB, Informix, JDBC, MySQL, Mysql2, ODBC, OpenBase, Oracle, PostgreSQL, SQLite3, Swift, and TinyTDS.

Resources

- Website
- Blog
- Source code
- Bug tracking
- Google group
- RDoc

To check out the source code:

```
git clone git://github.com/jeremyevans/sequel.git
```

Contact

If you have any comments or suggestions please post to the Google group.

Installation

```
sudo gem install sequel
```

A Short Example

```
require 'rubygems'
require 'sequel'
DB = Sequel.sqlite # memory database

DB.create_table :items do
  primary_key :id
  String :name
  Float :price
end

items = DB[:items] # Create a dataset

# Populate the table
items.insert(:name => 'abc', :price => rand * 100)
items.insert(:name => 'def', :price => rand * 100)
items.insert(:name => 'ghi', :price => rand * 100)

# Print out the number of records
puts "Item count: #{items.count}"

# Print out the average price
puts "The average price is: #{items.avg(:price)}"
```

The Sequel Console

Sequel includes an IRB console for quick access to databases. You can use it like this:

```
sequel sqlite://test.db # test.db in current directory
```

You get an IRB session with the database object stored in DB.

An Introduction

Sequel is designed to take the hassle away from connecting to databases and manipulating them. Sequel deals with all the boring stuff like maintaining connections, formatting SQL correctly and fetching records so you can concentrate on your application.

Sequel uses the concept of datasets to retrieve data. A Dataset object encapsulates an SQL query and supports chainability, letting you fetch data using a convenient Ruby DSL that is both concise and flexible.

For example, the following one-liner returns the average GDP for countries in the middle east region:

```
DB[:countries].filter(:region => 'Middle East').avg(:GDP)
```

Which is equivalent to:

```
SELECT avg(GDP) FROM countries WHERE region = 'Middle East'
```

Since datasets retrieve records only when needed, they can be stored and later reused. Records are fetched as hashes (or custom model objects), and are accessed using an `Enumerable` interface:

```
middle_east = DB[:countries].filter(:region => 'Middle East')
middle_east.order(:name).each{|r| puts r[:name]}
```

Sequel also offers convenience methods for extracting data from Datasets, such as an extended `map` method:

```
middle_east.map(:name) #=> ['Egypt', 'Greece', 'Israel', ...]
```

Or getting results as a hash via `to_hash`, with one column as key and another as value:

```
middle_east.to_hash(:name, :area) #=> {'Israel' => 20000, 'Greece' => 120000, ...}
```

Getting Started

Connecting to a database

To connect to a database you simply provide `Sequel.connect` with a URL:

```
require 'sequel'
DB = Sequel.connect('sqlite://blog.db')
```

The connection URL can also include such stuff as the user name, password, and port:

```
DB = Sequel.connect('postgres://user:password@host:port/database_name')
```

You can also specify optional parameters, such as the connection pool size, or loggers for logging SQL queries:

```
DB = Sequel.connect("postgres://user:password@host:port/database_name",
```

```
  :max_connections => 10, :logger => Logger.new('log/db.log'))
```

You can specify a block to connect, which will disconnect from the database after it completes:

```
Sequel.connect('postgres://user:password@host:port/database_name'){|db| db[:posts].delete}
```

Arbitrary SQL queries

You can execute arbitrary SQL code using `Database#run`:

```
DB.run("create table t (a text, b text)")
DB.run("insert into t values ('a', 'b')")
```

You can also create datasets based on raw SQL:

```
dataset = DB['select id from items']
dataset.count # will return the number of records in the result set
dataset.map(:id) # will return an array containing all values of the id column in the result set
```

You can also fetch records with raw SQL through the dataset:

```
DB['select * from items'].each do |row|
  p row
end
```

You can use placeholders in your SQL string as well:

```
name = 'Jim'
DB['select * from items where name = ?', name].each do |row|
  p row
end
```

Getting Dataset Instances

Datasets are the primary way records are retrieved and manipulated. They are generally created via the `Database#from` or `Database#[]` methods:

```
posts = DB.from(:posts)
posts = DB[:posts] # same
```

Datasets will only fetch records when you tell them to. They can be manipulated to filter records, change ordering, join tables, etc..

Retrieving Records

You can retrieve all records by using the `all` method:

```
posts.all
```

```
# SELECT * FROM posts
```

The all method returns an array of hashes, where each hash corresponds to a record.

You can also iterate through records one at a time using `each`:

```
posts.each{|row| p row}
```

Or perform more advanced stuff:

```
names_and_dates = posts.map{|r| [r[:name], r[:date]]}
old_posts, recent_posts = posts.partition{|r| r[:date] < Date.today - 7}
```

You can also retrieve the first record in a dataset:

```
posts.first
# SELECT * FROM posts LIMIT 1
```

Or retrieve a single record with a specific value:

```
posts[:id => 1]
# SELECT * FROM posts WHERE id = 1 LIMIT 1
```

If the dataset is ordered, you can also ask for the last record:

```
posts.order(:stamp).last
# SELECT * FROM posts ORDER BY stamp DESC LIMIT 1
```

Filtering Records

An easy way to filter records is to provide a hash of values to match to `filter`:

```
my_posts = posts.filter(:category => 'ruby', :author => 'david')
# WHERE category = 'ruby' AND author = 'david'
```

You can also specify ranges:

```
my_posts = posts.filter(:stamp => (Date.today - 14)..(Date.today - 7))
# WHERE stamp >= '2010-06-30' AND stamp <= '2010-07-07'
```

Or arrays of values:

```
my_posts = posts.filter(:category => ['ruby', 'postgres', 'linux'])
# WHERE category IN ('ruby', 'postgres', 'linux')
```

Sequel also accepts expressions:

```
my_posts = posts.filter{stamp > Date.today << 1}
# WHERE stamp > '2010-06-14'
```

Some adapters will also let you specify Regexps:

```
my_posts = posts.filter(:category => /ruby/i)
# WHERE category ~* 'ruby'
```

You can also use an inverse filter via `exclude`:

```
my_posts = posts.exclude(:category => ['ruby', 'postgres', 'linux'])
# WHERE category NOT IN ('ruby', 'postgres', 'linux')
```

You can also specify a custom WHERE clause using a string:

```
posts.filter('stamp IS NOT NULL')
# WHERE stamp IS NOT NULL
```

You can use parameters in your string, as well:

```
author_name = 'JKR'
posts.filter('(stamp < ?) AND (author != ?)', Date.today - 3, author_name)
# WHERE (stamp < '2010-07-11') AND (author != 'JKR')
posts.filter{(stamp < Date.today - 3) & ~{:author => author_name}} # same as above
```

Datasets can also be used as subqueries:

```
DB[:items].filter('price > ?', DB[:items].select{avg(price) + 100})
# WHERE price > (SELECT avg(price) + 100 FROM items)
```

After filtering you can retrieve the matching records by using any of the retrieval methods:

```
my_posts.each{|row| p row}
```

See the doc/dataset_filtering.rdoc file for more details.

Summarizing Records

Counting records is easy using `count`:

```
posts.filter(:category.like('%ruby%')).count
# SELECT COUNT(*) FROM posts WHERE category LIKE '%ruby%'
```

And you can also query maximum/minimum values via `max` and `min`:

```
max = DB[:history].max(:value)
# SELECT max(value) FROM history
min = DB[:history].min(:value)
# SELECT min(value) FROM history
```

Or calculate a sum or average via `sum` and `avg`:

```
sum = DB[:items].sum(:price)
```

```
# SELECT sum(price) FROM items
avg = DB[:items].avg(:price)
# SELECT avg(price) FROM items
```

Ordering Records

Ordering datasets is simple using `order`:

```
posts.order(:stamp)
# ORDER BY stamp
posts.order(:stamp, :name)
# ORDER BY stamp, name
```

Chaining `order` doesn't work the same as `filter`:

```
posts.order(:stamp).order(:name)
# ORDER BY name
```

The `order_append` method chains this way, though:

```
posts.order(:stamp).order_append(:name)
# ORDER BY stamp, name
```

You can also specify descending order:

```
posts.order(:stamp.desc)
# ORDER BY stamp DESC
```

Selecting Columns

Selecting specific columns to be returned is also simple using `select`:

```
posts.select(:stamp)
# SELECT stamp FROM posts
posts.select(:stamp, :name)
# SELECT stamp, name FROM posts
```

Chaining `select` works like `order`, not `filter`:

```
posts.select(:stamp).select(:name)
# SELECT name FROM posts
```

As you might expect, there is an `order_append` equivalent for `select` called `select_append`:

```
posts.select(:stamp).select_append(:name)
# SELECT stamp, name FROM posts
```

Deleting Records

Deleting records from the table is done with `delete`:

```
posts.filter('stamp < ?', Date.today - 3).delete
# DELETE FROM posts WHERE stamp < '2010-07-11'
```

Be very careful when deleting, as `delete` affects all rows in the dataset. `filter` first, `delete` second, unless you want to empty the table:

```
# DO THIS:
posts.filter('stamp < ?', Date.today - 7).delete
# NOT THIS:
posts.delete.filter('stamp < ?', Date.today - 7)
```

Inserting Records

Inserting records into the table is done with `insert`:

```
posts.insert(:category => 'ruby', :author => 'david')
# INSERT INTO posts (category, author) VALUES ('ruby', 'david')
```

Updating Records

Updating records in the table is done with `update`:

```
posts.filter('stamp < ?', Date.today - 7).update(:state => 'archived')
# UPDATE posts SET state = 'archived' WHERE stamp < '2010-07-07'
```

You can reference table columns when choosing what values to set:

```
posts.filter{|o| o.stamp < Date.today - 7}.update(:backup_number => :backup_number + 1)
# UPDATE posts SET backup_number = backup_number + 1 WHERE stamp < '2010-07-07'
```

As with `delete`, `update` affects all rows in the dataset, so `filter` first, `update` second, unless you want to update all rows:

```
# DO THIS:
posts.filter('stamp < ?', Date.today - 7).update(:state => 'archived')
# NOT THIS:
posts.update(:state => 'archived').filter('stamp < ?', Date.today - 7)
```

Transactions

You can wrap some code in a database transaction using the `Database#transaction` method:

```
DB.transaction do
  posts.insert(:category => 'ruby', :author => 'david')
  posts.filter('stamp < ?', Date.today - 7).update(:state => 'archived')
end
```

If the block does not raise an exception, the transaction will be committed. If the block does raise an exception, the transaction will be rolled back, and the exception will be reraised. If you want to rollback the transaction and not raise an exception outside the block, you can raise the `Sequel::Rollback` exception inside the block:

```
DB.transaction do
  posts.insert(:category => 'ruby', :author => 'david')
  if posts.filter('stamp < ?', Date.today - 7).update(:state => 'archived') == 0
    raise Sequel::Rollback
  end
end
```

Joining Tables

Sequel makes it easy to join tables:

```
order_items = DB[:items].join(:order_items, :item_id => :id).
  filter(:order_items__order_id => 1234)
# SELECT * FROM items INNER JOIN order_items
# ON order_items.item_id = items.id
# WHERE order_items.order_id = 1234
```

You can then do anything you like with the dataset:

```
order_total = order_items.sum(:price)
# SELECT sum(price) FROM items INNER JOIN order_items
# ON order_items.item_id = items.id
# WHERE order_items.order_id = 1234
```

Graphing Datasets

When retrieving records from joined datasets, you get the results in a single hash, which is subject to clobbering if you have columns with the same name in multiple tables:

```
DB[:items].join(:order_items, :item_id => :id).first
=> {:id=>order_items.id, :item_id=>order_items.item_id}
```

Using `graph`, you can split the result hashes into subhashes, one per join:

```
DB[:items].graph(:order_items, :item_id => :id).first
=> {:items=>{:id=>items.id}, :order_items=>{:id=>order_items.id, :item_id=>order_items.item_id}}
```

Column references in Sequel

Sequel expects column names to be specified using symbols. In addition, returned hashes always use symbols as their keys. This allows you to freely mix

literal values and column references in many cases. For example, the two following lines produce equivalent SQL:

```
items.filter(:x => 1)
# SELECT * FROM items WHERE (x = 1)
items.filter(1 => :x)
# SELECT * FROM items WHERE (1 = x)"
```

Ruby strings are generally treated as SQL strings:

```
items.filter(:x => 'x')
# SELECT * FROM items WHERE (x = 'x')
```

Qualifying column names

Column references can be qualified by using the double underscore special notation `:table__column`:

```
items.literal(:items__price)
# items.price
```

Another way to qualify columns is to use the `qualify` method:

```
items.literal(:price.qualify(:items))
# items.price
```

Column aliases

You can also alias columns by using the triple undersecore special notation `:column___alias` or `:table__column___alias`:

```
items.literal(:price___p)
# price AS p
items.literal(:items__price___p)
# items.price AS p
```

Another way to alias columns is to use the `as` method:

```
items.literal(:price.as(:p))
# price AS p
```

Sequel Models

A model class wraps a dataset, and an instance of that class wraps a single record in the dataset.

Model classes are defined as regular Ruby classes inheriting from `Sequel::Model`:

```
DB = Sequel.connect('sqlite://blog.db')
```

```
class Post < Sequel::Model
end
```

Sequel model classes assume that the table name is an underscored plural of the class name:

```
Post.table_name #=> :posts
```

You can explicitly set the table name or even the dataset used:

```
class Post < Sequel::Model(:my_posts)
end
# or:
Post.set_dataset :my_posts
```

If you call `set_dataset` with a symbol, it assumes you are referring to the table with the same name. You can also call it with a dataset, which will set the defaults for all retrievals for that model:

```
Post.set_dataset DB[:my_posts].filter(:category => 'ruby')
Post.set_dataset DB[:my_posts].select(:id, :name).order(:date)
```

Model instances

Model instances are identified by a primary key. In most cases, Sequel can query the database to determine the primary key, but if not, it defaults to using `:id`. The `Model.[]` method can be used to fetch records by their primary key:

```
post = Post[123]
```

The `pk` method is used to retrieve the record's primary key value:

```
post.pk #=> 123
```

Sequel models allow you to use any column as a primary key, and even composite keys made from multiple columns:

```
class Post < Sequel::Model
  set_primary_key [:category, :title]
end
post = Post['ruby', 'hello world']
post.pk #=> ['ruby', 'hello world']
```

You can also define a model class that does not have a primary key via `no_primary_key`, but then you lose the ability to easily update and delete records:

```
Post.no_primary_key
```

A single model instance can also be fetched by specifying a condition:

```
post = Post[:title => 'hello world']
post = Post.first{num_comments < 10}
```

Iterating over records

A model class lets you iterate over subsets of records by proxying many methods to the underlying dataset. This means that you can use most of the `Dataset` API to create customized queries that return model instances, e.g.:

```
Post.filter(:category => 'ruby').each{|post| p post}
```

You can also manipulate the records in the dataset:

```
Post.filter{num_comments < 7}.delete
Post.filter(:title.like(/ruby/)).update(:category => 'ruby')
```

Accessing record values

A model instance stores its values as a hash with column symbol keys, which you can access directly via the `values` method:

```
post.values #=> {:id => 123, :category => 'ruby', :title => 'hello world'}
```

You can read the record values as object attributes, assuming the attribute names are valid columns in the model's dataset:

```
post.id #=> 123
post.title #=> 'hello world'
```

If the record's attributes names are not valid columns in the model's dataset (maybe because you used `select_append` to add a computed value column), you can use `Model#[]` to access the values:

```
post[:id] #=> 123
post[:title] #=> 'hello world'
```

You can also modify record values using attribute setters or the `set` method:

```
post.title = 'hey there'
# or
post.set(:title=>'hey there')
```

That will just change the value for the object, it will not update the row in the database. To update the database row, call the `save` method:

```
post.save
```

You can modify record values and save the changes to the object in a single method call using the `update` method:

```
post.update(:title => 'hey there')
```

Creating new records

New records can be created by calling `Model.create`:

```
post = Post.create(:title => 'hello world')
```

Another way is to construct a new instance and save it later:

```
post = Post.new
post.title = 'hello world'
post.save
```

You can also supply a block to `Model.new` and `Model.create`:

```
post = Post.new do |p|
  p.title = 'hello world'
end
post = Post.create{|p| p.title = 'hello world'}
```

Hooks

You can execute custom code when creating, updating, or deleting records by defining hook methods. The `before_create` and `after_create` hook methods wrap record creation. The `before_update` and `after_update` hook methods wrap record updating. The `before_save` and `after_save` hook methods wrap record creation and updating. The `before_destroy` and `after_destroy` hook methods wrap destruction. The `before_validation` and `after_validation` hook methods wrap validation. Example:

```
class Post < Sequel::Model
  def after_create
    super
    author.increase_post_count
  end
  def after_destroy
    super
    author.decrease_post_count
  end
end
```

Note the use of `super` if you define your own hook methods. Almost all `Sequel::Model` class and instance methods (not just hook methods) can be overridden safely, but you have to make sure to call `super` when doing so, otherwise you risk breaking things.

For the example above, you should probably use a database trigger if you can. Hooks can be used for data integrity, but they will only enforce that integrity when you are modifying the database through model instances. If you plan on

allowing any other access to the database, it's best to use database triggers and constraints for data integrity.

Deleting records

You can delete individual records by calling `delete` or `destroy`. The only difference between the two methods is that `destroy` invokes `before_destroy` and `after_destroy` hook methods, while `delete` does not:

```
post.delete # => bypasses hooks
post.destroy # => runs hooks
```

Records can also be deleted en-masse by calling `Model.delete` and `Model.destroy`. As stated above, you can specify filters for the deleted records:

```
Post.filter(:category => 32).delete # => bypasses hooks
Post.filter(:category => 32).destroy # => runs hooks
```

Please note that if `Model.destroy` is called, each record is deleted separately, but `Model.delete` deletes all matching records with a single SQL query.

Associations

Associations are used in order to specify relationships between model classes that reflect relationships between tables in the database, which are usually specified using foreign keys. You specify model associations via the `many_to_one`, `one_to_one`, `one_to_many`, and `many_to_many` class methods:

```
class Post < Sequel::Model
  many_to_one :author
  one_to_many :comments
  many_to_many :tags
end
```

`many_to_one` and `one_to_one` create a getter and setter for each model object:

```
post = Post.create(:name => 'hi!')
post.author = Author[:name => 'Sharon']
post.author
```

`one_to_many` and `many_to_many` create a getter method, a method for adding an object to the association, a method for removing an object from the association, and a method for removing all associated objects from the association:

```
post = Post.create(:name => 'hi!')
post.comments
comment = Comment.create(:text=>'hi')
post.add_comment(comment)
post.remove_comment(comment)
```

```
post.remove_all_comments

tag = Tag.create(:tag=>'interesting')
post.add_tag(tag)
post.remove_tag(tag)
post.remove_all_tags
```

Note that the remove_* and remove_all_* methods do not delete the object from the database, they merely disassociate the associated object from the receiver.

All associations add a dataset method that can be used to further filter or reorder the returned objects, or modify all of them:

```
# Delete all of this post's comments from the database
post.comments_dataset.destroy
# Return all tags related to this post with no subscribers, ordered by the tag's name
post.tags_dataset.filter(:subscribers=>0).order(:name).all
```

Eager Loading

Associations can be eagerly loaded via `eager` and the `:eager` association option. Eager loading is used when loading a group of objects. It loads all associated objects for all of the current objects in one query, instead of using a separate query to get the associated objects for each current object. Eager loading requires that you retrieve all model objects at once via `all` (instead of individually by `each`). Eager loading can be cascaded, loading association's associated objects.

```
class Person < Sequel::Model
  one_to_many :posts, :eager=>[:tags]
end
class Post < Sequel::Model
  many_to_one :person
  one_to_many :replies
  many_to_many :tags
end

class Tag < Sequel::Model
  many_to_many :posts
  many_to_many :replies
end

class Reply < Sequel::Model
  many_to_one :person
  many_to_one :post
  many_to_many :tags
end

# Eager loading via .eager
Post.eager(:person).all
```

```
# eager is a dataset method, so it works with filters/orders/limits/etc.
Post.filter{topic > 'M'}.order(:date).limit(5).eager(:person).all

person = Person.first
# Eager loading via :eager (will eagerly load the tags for this person's posts)
person.posts

# These are equivalent
Post.eager(:person, :tags).all
Post.eager(:person).eager(:tags).all

# Cascading via .eager
Tag.eager(:posts=>:replies).all

# Will also grab all associated posts' tags (because of :eager)
Reply.eager(:person=>:posts).all

# No depth limit (other than memory/stack), and will also grab posts' tags
# Loads all people, their posts, their posts' tags, replies to those posts,
# the person for each reply, the tag for each reply, and all posts and
# replies that have that tag.  Uses a total of 8 queries.
Person.eager(:posts=>{:replies=>[:person, {:tags=>[:posts, :replies]}]}).all
```

In addition to using `eager`, you can also use `eager_graph`, which will use a single query to get the object and all associated objects. This may be necessary if you want to filter or order the result set based on columns in associated tables. It works with cascading as well, the API is very similar. Note that using `eager_graph` to eagerly load multiple `*_to_many` associations will cause the result set to be a cartesian product, so you should be very careful with your filters when using it in that case.

You can dynamically customize the eagerly loaded dataset by using using a proc. This proc is passed the dataset used for eager loading, and should return a modified copy of that dataset:

```
# Eagerly load only replies containing 'foo'
Post.eager(:replies=>proc{|ds| ds.filter(text.like('%foo%'))}).all
```

This also works when using `eager_graph`, in which case the proc is called with dataset to graph into the current dataset:

```
Post.eager_graph(:replies=>proc{|ds| ds.filter(text.like('%foo%'))}).all
```

You can dynamically customize eager loads for both `eager` and `eager_graph` while also cascading, by making the value a single entry hash with the proc as a key, and the cascaded associations as the value:

```
# Eagerly load only replies containing 'foo', and the person and tags for those replies
Post.eager(:replies=>{proc{|ds| ds.filter(text.like('%foo%'))}=>[:person, :tags]}).all
```

Extending the underlying dataset

The obvious way to add table-wide logic is to define class methods to the model class definition. That way you can define subsets of the underlying dataset, change the ordering, or perform actions on multiple records:

```
class Post < Sequel::Model
  def self.posts_with_few_comments
    filter{num_comments < 30}
  end
  def self.clean_posts_with_few_comments
    posts_with_few_comments.delete
  end
end
```

You can also implement table-wide logic by defining methods on the dataset using `def_dataset_method`:

```
class Post < Sequel::Model
  def_dataset_method(:posts_with_few_comments) do
    filter{num_comments < 30}
  end
  def_dataset_method(:clean_posts_with_few_comments) do
    posts_with_few_comments.delete
  end
end
```

This is the recommended way of implementing table-wide operations, and allows you to have access to your model API from filtered datasets as well:

```
Post.filter(:category => 'ruby').clean_posts_with_few_comments
```

Sequel models also provide a `subset` class method that creates a dataset method with a simple filter:

```
class Post < Sequel::Model
  subset(:posts_with_few_comments){num_comments < 30}
  subset :invisible, ~:visible
end
```

Model Validations

You can define a `validate` method for your model, which `save` will check before attempting to save the model in the database. If an attribute of the model isn't valid, you should add a error message for that attribute to the model object's `errors`. If an object has any errors added by the validate method, `save` will raise an error or return false depending on how it is configured (the `raise_on_save_failure` flag).

```
class Post < Sequel::Model
  def validate
    super
    errors.add(:name, "can't be empty") if name.empty?
    errors.add(:written_on, "should be in the past") if written_on >= Time.now
  end
end
```

Hanna RDoc template