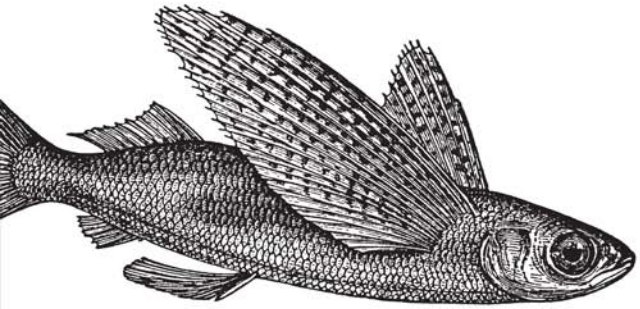
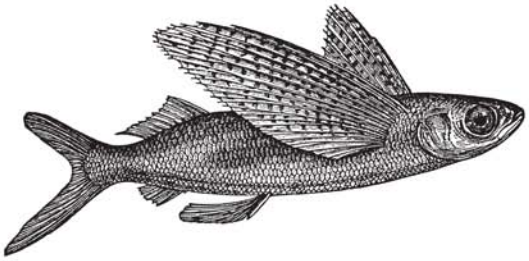
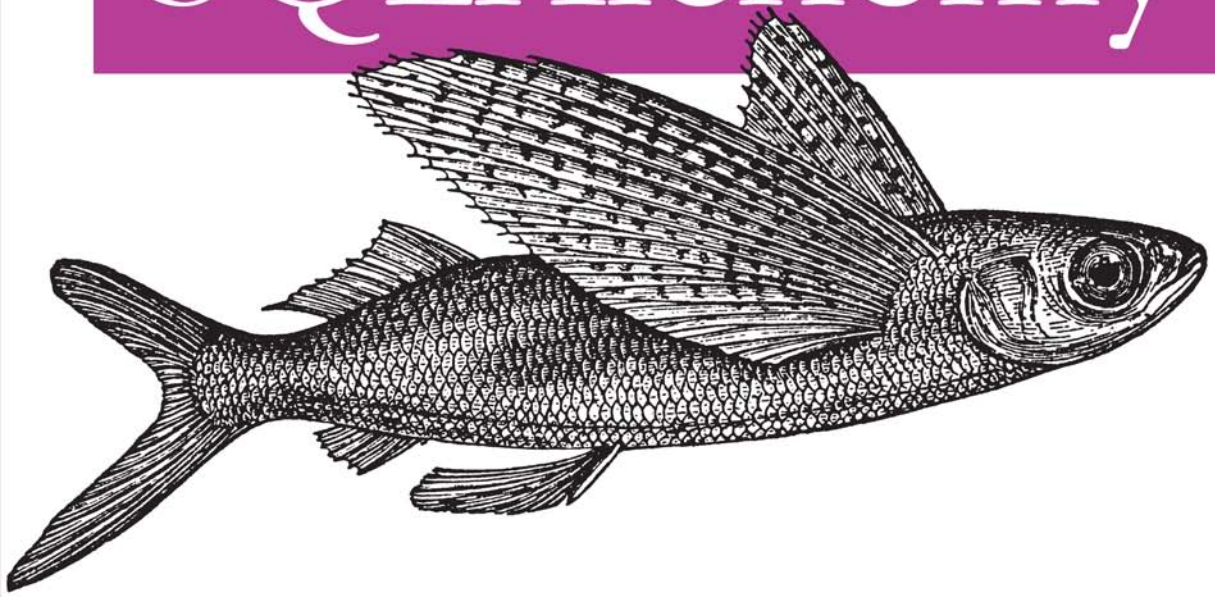
Mapping Python to Databases





Essential

SQLAlchemy



O'REILLY®

Rick Copeland

Databases

O'REILLY®

Essential SQLAlchemy

Now Python developers can easily access relational databases such as Oracle, DB2, and MySQL. *Essential SQLAlchemy* walks you through simple queries, demonstrates how to create database applications, explains how to connect to multiple databases simultaneously with the same metadata, and more. With this practical guide, you'll learn how the SQLAlchemy open source code library lets you map objects to database tables without substantially changing your existing Python code. You'll also learn how to:

- Create custom types for your schema and discover when to use them
- Create objects, save them to a session, and flush them to the database
- Run queries, updates, and deletes with SQLAlchemy's SQL expression language
- Build an object mapper and learn why it's different from active record patterns used in other ORMs
- Provide a declarative, active record pattern for use with SQLAlchemy via the Elixir extension
- Use the SQLSoup extension to provide an automatic metadata and object model based on database reflection
- Use SQLAlchemy to model object-oriented inheritance

This book gives you an objective look at SQLAlchemy from a developer's viewpoint rather than from an advocate's description. It's exactly what you need to quickly get up to speed with this code library.

Rick Copeland is a senior software engineer with retail analytics firm Predictix, LLC, where he uses SQLAlchemy extensively, primarily for web application development.

"SQLAlchemy may be the best way to use a relational database from Python, and this book will certainly help you exploit SQLAlchemy's power, if you start with a strong understanding of Python and SQL."

—Alex Martelli, über tech lead, Google Inc., and author of *Python in a Nutshell* (O'Reilly)

"Essential SQLAlchemy *provides much-needed guidance for programmers who are familiar with SQL but new to database mappers."*

—Liza Daly, senior software engineer, ifactory.com, and author of *Next-Generation Web Frameworks in Python* (O'Reilly)

www.oreilly.com

US $34.99 **CAN $34.99**
ISBN: 978-0-596-51614-7

Essential SQLAlchemy

Essential SQLAlchemy

Rick Copeland

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Essential SQLAlchemy
by Rick Copeland

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mary E. Treseler
Copy Editor: Genevieve d'Entremont
Production Editor: Sumita Mukherji
Proofreader: Sumita Mukherji

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Jessamyn Read

Printing History:
June 2008: First Edition

ISBN: 978-0-596-51614-7

[M]

1210573118

Table of Contents

Preface ..... **vii**

1. Introduction to SQLAlchemy **1**
What Is SQLAlch 1
The Object/Relational "Impedance Mismatch" 4
SQLAlchemy Philosophy 7
SQLAlchemy Architecture 10

2. Getting Started **21**
Installing SQLAlchemy 21
SQLAlchemy Tutorial 24

3. Engines and MetaData **33**
Engines and Connectables 33
MetaData 39

4. SQLAlchemy Type Engines **59**
Type System Overview 59
Built-in Types 59
Application-Specific Custom Types 63

5. Running Queries and Updates **67**
Inserts, Updates, and Deletes 67
Queries 72

6. Building an Object Mapper **93**
Introduction to ORMs 93
Declaring Object Mappers 95
Declaring Relationships Between Mappers 108
Extending Mappers 120
ORM Partitioning Strategies 122

7. Querying and Updating at the ORM Level 127
The SQLAlchemy ORM Session Object 127
Querying at the ORM Level 139
Contextual or Thread-Local Sessions 153

8. Inheritance Mapping 157
Overview of Inheritance Mapping 157
Single Table Inheritance Mapping 158
Concrete Table Inheritance Mapping 161
Joined Table Inheritance Mapping 163
Relations and Inheritance 168

9. Elixir: A Declarative Extension to SQLAlchemy 171
Introduction to Elixir 171
Installing Elixir 174
Using Elixir 174
Elixir Extensions 184

10. SqlSoup: An Automatic Mapper for SQLAlchemy 189
Introduction to SqlSoup 189
Using SqlSoup for ORM-Style Queries and Updates 191
Using SqlSoup for SQL-Level Inserts, Updates, and Deletes 195
When to Use SqlSoup Versus Elixir Versus "Bare" SQLAlchemy 195

11. Other SQLAlchemy Extensions 199
Association Proxy 199
Ordering List 203
Deprecated Extensions 205

Index 207

Preface

If you're an application programmer you've probably run into a relational database at some point in your professional career. Whether you're writing enterprise client-server applications or building the next Web 2.0 killer application, you need someplace to put the persistent data for your application, and relational databases, accessed via SQL, are some of the most common places to put that data.

SQL is a powerful language for querying and manipulating data in a database, but sometimes it's tough to integrate it with the rest of your application. You may have used some language that tries to merge SQL syntax into your application's programming language, such as Oracle's Pro*C/C++ precompiler, or you may have used string manipulation to generate queries to run over an ODBC interface. If you're a Python programmer, you may have used a DB-API module. But there is a better way.

This book is about a very powerful and flexible Python library named SQLAlchemy that bridges the gap between relational databases and traditional object-oriented programming. While SQLAlchemy allows you to "drop down" into raw SQL to execute your queries, it encourages higher-level thinking through a "pythonic" approach to database queries and updates. It supplies the tools that let you map your application's classes and objects onto database tables once and then "forget about it," or return to your model again and again to fine-tune performance.

SQLAlchemy is powerful and flexible, but it can also be a little daunting. SQLAlchemy tutorials expose only a fraction of what's available in this excellent library, and though the online documentation is extensive, it is often better as a reference than a way to learn the library initially. This book is meant as both a learning tool and a handy reference for when you're in "implementation mode," and need an answer *fast*.

This book covers the 0.4 release series of conservatively versioned SQLAlchemy.

Audience

First of all, this book is intended for those who want to learn more about how to use relational databases with their Python programs, or have heard about SQLAlchemy and want more information on it. Having said that, to get the most out of this book,

the reader should have intermediate-to-advanced Python skills and at least moderate exposure to SQL databases. SQLAlchemy provides support for many advanced SQL constructs, so the experienced DBA will also find plenty of information here.

The beginning Python or database programmer would probably be best served by reading a Python book such as *Learning Python* and/or a SQL book such as *Learning SQL*, either prior to this book or as a reference to read in parallel with this book.

Assumptions This Book Makes

This book assumes basic knowledge about Python syntax and semantics, particularly versions 2.4 and later. In particular, the reader should be familiar with object-oriented programming in Python, as a large component of SQLAlchemy is devoted entirely to supporting this programming style. The reader should also know basic SQL syntax and relational theory, as this book assumes familiarity with the SQL concepts of defining schemas, tables, SELECTs, INSERTs, UPDATEs, and DELETEs.

Contents of this Book

Chapter 1, Introduction to SQLAlchemy
: This chapter takes you on a whirlwind tour through the main components of SQLAlchemy. It demonstrates connecting to the database, building up SQL statements, and mapping simple objects to the database. It also describes SQLAlchemy's philosophy of letting tables be tables and letting classes be classes.

Chapter 2, Getting Started
: This chapter walks you through installing SQLAlchemy using *easy_install*. It shows you how to create a simple database using SQLite, and walks though some simple queries against a sample database to to illustrate the use of the Engine and the SQL expression language.

Chapter 3, Engines and MetaData
: This chapter describes the various engines (methods of connecting to database servers) available for use with SQLAlchemy, including the connection parameters they support. It then describes the `MetaData` object, which is where SQLAlchemy stores information about your database's schema, and how to manipulate `MetaData` objects.

Chapter 4, SQLAlchemy Type Engines
: This chapter describes the way that SQLAlchemy uses its built-in types. It also shows how you can create custom types to be used in your schema. You will learn the requirements for creating custom types as well as the cases where it is useful to use custom rather than built-in types.

Chapter 5, Running Queries and Updates
: This chapter illustrates how to perform inserts, updates, and deletes. It covers result set objects, retrieving partial results, and using SQL functions to aggregate and sort data in the database server.

Chapter 6, Building an Object Mapper
: This chapter describes the object-relational mapper (ORM) used in SQLAlchemy. It describes the differences between the object mapper pattern (used in SQLAlchemy) and the active record pattern used in other ORMs. It then describes how to set up a mapper, and how the mapper maps your tables by default. You will also learn how to override the default mapping and how to specify various relationships between tables.

Chapter 7, Querying and Updating at the ORM Level
: This chapter shows how to create objects, save them to a session, and flush them to the database. You will learn how `Session` and `Query` objects are defined, their methods, and how to use them to insert, update, retrieve, and delete data from the database at the ORM level. You will learn how to use result set mapping to populate objects from a non-ORM query and when it should be used.

Chapter 8, Inheritance Mapping
: This chapter describes how to use SQLAlchemy to model object-oriented inheritance. The various ways of modeling inheritance in the relational model are described, as well has the support SQLAlchemy provides for each.

Chapter 9, Elixir: A Declarative Extension to SQLAlchemy
: This chapter describes the Elixir extension to SQLAlchemy, which provides a declarative, active record pattern for use with SQLAlchemy. You will learn how to use Elixir extensions such as `acts_as_versioned` to create auxilliary tables automatically, and when Elixir is appropriate instead of "bare" SQLAlchemy.

Chapter 10, SqlSoup: An Automatic Mapper for SQLAlchemy
: This chapter introduces the SQLSoup extension, which provides an automatic metadata and object model based on database reflection. You will learn how to use SQLSoup to query the database with a minimum of setup, and learn the pros and cons of such an approach.

Chapter 11, Other SQLAlchemy Extensions
: This chapter covers other, less extensive, extensions to SQLAlchemy. This chapter describes the extensions that are currently used in the 0.4 release series of SQLAlchemy, as well as briefly describing deprecated extensions, and the functionality in SQLAlchemy that supplants them.

Conventions Used in This Book

The following typographical conventions are used in this book:

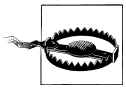
Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, the contents of files, or the output from commands.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

ALL CAPS
: Shows SQL keywords and queries.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Essential SQLAlchemy* by Rick Copeland. Copyright 2008 Richard D. Copeland, Jr., 978-0-596-51614-7."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596516147

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

Many thanks go to Tatiana Apandi, TECHNICAL REVIEWERS HERE for their critical pre-publication feedback, without whom this book would have undoubtedly had many technical snafus.

My appreciation goes out to Noah Gift, whose recommendation led to this book being written in the first place. I still remember how his phone call started: "You know SQLAlchemy, right?..."

Thanks to my employer, Predictix, for allowing me the time and energy to finish the book, and to my co-workers for being unwitting guinea pigs for many of the ideas and techniques in this book.

Finally, my heartfelt gratitude goes to my beloved wife Nancy, whose support in the presence of a husband glued to the computer was truly the fuel that allowed this book to be written at all.

CHAPTER 1

Introduction to SQLAlchemy

What Is SQLAlch

SQLALchemy is a Python Library created by Mike Bayer to provide a high-level, Pythonic (idiomatically Python) interface to relational databases such as Oracle, DB2, MySQL, PostgreSQL, and SQLite. SQLAlchemy attempts to be unobtrusive to your Python code, allowing you to map plain old Python objects (POPOs) to database tables without substantially changing your existing Python code. SQLAlchemy includes both a database server–independent SQL expression language and an object-relational mapper (ORM) that lets you use SQL to persist your application objects automatically. This chapter will introduce you to SQLAlchemy, illustrating some of its more powerful features. Later chapters will provide more depth for the topics covered here.

If you have used lower-level database interfaces with Python, such as the DB-API, you may be used to writing code such as the following to save your objects to the database:

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

Although this code gets the job done, it is verbose, error-prone, and tedious to write. Using string manipulation to build up a query as done here can lead to various logical errors and vulnerabilities such as opening your application up to SQL injection attacks. Generating the string to be executed by your database server verbatim also ties your code to the particular DB-API driver you are currently using, making migration to a different database server difficult. For instance, if we wished to migrate the previous example to the Oracle DB-API driver, we would need to write:

```
sql="INSERT INTO user(user_name, password) VALUES (:1, :2)"
cursor = conn.cursor()
cursor.execute(sql, 'rick', 'parrot')
```

SQL Injection Attacks

SQL injection is a type of programming error where carefully crafted user input can cause your application to execute arbitrary SQL code. For instance, suppose that the DB-API code in the earlier listing had been written as follows:

```
sql="INSERT INTO user(user_name, password) VALUES ('%s', '%s')"
cursor = conn.cursor()
cursor.execute(sql % (user_name, password))
```

In most cases, this code will work. For instance, with the `user_name` and `password` variables just shown, the SQL that would be executed is `INSERT INTO user(user_name, password) VALUES ('rick', 'parrot')`. A user could, however, supply a maliciously crafted password: `parrot'); DELETE FROM user; --`. In this case, the SQL executed is `INSERT INTO user(user_name, password) VALUES ('rick', 'parrot'); DELETE FROM user; --'`, which would probably delete all users from your database. The use of bind parameters (as in the first example in the text) is an effective defense against SQL injection, but as long as you are manipulating strings directly, there is always the possibility of introducting a SQL injection vulnerability into your code.

In the SQLAlchemy SQL expression language, you could write the following instead:

```
statement = user_table.insert(user_name='rick', password='parrot')
statement.execute()
```

To migrate this code to Oracle, you would write, well, exactly the same thing.

SQLAlchemy also allows you to write SQL queries using a Pythonic expression-builder. For instance, to retrieve all the users created in 2007, you would write:

```
statement = user_table.select(and_(
    user_table.c.created >= date(2007,1,1),
    user_table.c.created <  date(2008,1,1))
result = statement.execute()
```

In order to use the SQL expression language, you need to provide SQLAlchemy with information about your database schema. For instance, if you are using the user table mentioned previously, your schema definition might be the following:

```
metadata=MetaData('sqlite://') # use an in-memory SQLite database
user_table = Table(
    'tf_user', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_name', Unicode(16), unique=True, nullable=False),
Column('email_address', Unicode(255), unique=True, nullable=False),
    Column('password', Unicode(40), nullable=False),
    Column('first_name', Unicode(255), default=''),
    Column('last_name', Unicode(255), default=''),
    Column('created', DateTime, default=datetime.now))
```

If you would rather use an existing database schema definition, you still need to tell SQLAlchemy which tables you have, but SQLAlchemy can reflect the tables using the

database server's introspection capabilities. In this case, the schema definition reduces to the following:

```
users_table = Table('users', metadata, autoload=True)
```

Although the SQLAlchemy SQL expression language is quite powerful, it can still be tedious to manually specify the queries and updates necessary to work with your tables. To help with this problem, SQLAlchemy provides an ORM to automatically populate your Python objects from the database and to update the database based on changes to your Python objects. Using the ORM is as simple as writing your classes, defining your tables, and mapping your tables to your classes. In the case of the user table, you could perform a simple mapping via the following code:

```
class User(object): pass
mapper(User, user_table)
```

Notice that there is nothing particularly special about the `User` class defined here. It is used to create "plain old Python objects," or POPOs. All the magic of SQLAlchemy is performed by the mapper. Although the class definition just shown is empty, you may define your own methods and attributes on a mapped class. The mapper will create attributes corresponding to the column names in the mapped table as well as some private attributes used by SQLAlchemy internally. Once your table is mapped, you can use a `Session` object to populate your objects based on data in the user table and flush any changes you make to mapped objects to the database:

```
>>> Session = sessionmaker()
>>> session = Session()
>>>
>>> # Insert a user into the database
... u = User()
>>> u.user_name='rick'
>>> u.email_address='rick@foo.com'
>>> u.password='parrot'
>>> session.save(u)
>>>
>>> # Flush all changes to the session out to the database
... session.flush()
>>>
>>> query = session.query(User)
>>> # List all users
... list(query)
[<__main__.User object at 0x2abb96dae3d0>]
>>>
>>> # Get a particular user by primary key
... query.get(1)
<__main__.User object at 0x2abb96dae3d0>
>>>
>>> # Get a particular user by some other column
... query.get_by(user_name='rick')
<__main__.User object at 0x2abb96dae3d0>
>>>
>>> u = query.get_by(user_name='rick')
>>> u.password = 'foo'
```

```
>>> session.flush()
>>> query.get(1).password
'foo'
```

As you can see, SQLAlchemy makes persisting your objects simple and concise. You can also customize and extend the set of properties created by SQLAlchemy, allowing your objects to model, for instance, a many-to-many relationship with simple Python lists.

The Object/Relational "Impedance Mismatch"

Although a SQL database is a powerful and flexible modeling tool, it is not always a good match for the object-oriented programming style. SQL is good for some things, and object-oriented programming is good for others. This is sometimes referred to as the object/relational "impedance mismatch," and it is a problem that SQLAlchemy tries to address in the ORM. To illustrate the object/relational impedance mismatch, let's first look at how we might model a system in SQL, and then how we might model it in and object-oriented way.

SQL databases provide a powerful means for modeling data and allowing for arbitrary queries of that data. The model underlying SQL is the *relational model*. In the relational model, modeled items (*entities*) can have various attributes, and are related to other entities via *relationships*. These relationships can be one-to-one, one-to-many, many-to-many, or complex, multientity relationships. The SQL expression of the entity is the table, and relationships are expressed as foreign key constraints, possibly with the use of an auxiliary "join" table. For example, suppose we have a user permission system that has users who may belong to one or more groups. Groups may have one or more permissions. Our SQL to model such a system might be something like the following:

```
CREATE TABLE tf_user (
        id INTEGER NOT NULL,
        user_name VARCHAR(16) NOT NULL,
        email_address VARCHAR(255) NOT NULL,
        password VARCHAR(40) NOT NULL,
        first_name VARCHAR(255),
        last_name VARCHAR(255),
        created TIMESTAMP,
        PRIMARY KEY (id),
         UNIQUE (user_name),
         UNIQUE (email_address));
CREATE TABLE tf_group (
        id INTEGER NOT NULL,
        group_name VARCHAR(16) NOT NULL,
        PRIMARY KEY (id),
         UNIQUE (group_name));
CREATE TABLE tf_permission (
        id INTEGER NOT NULL,
        permission_name VARCHAR(16) NOT NULL,
        PRIMARY KEY (id),
         UNIQUE (permission_name));
```

```
-- Relate the user and group tables
CREATE TABLE user_group (
        user_id INTEGER,
        group_id INTEGER,
     PRIMARY KEY(user_id, group_id),
         FOREIGN KEY(user_id) REFERENCES tf_user (id),
         FOREIGN KEY(group_id) REFERENCES tf_group (id));
-- Relate the group and permission tables
CREATE TABLE group_permission (
        group_id INTEGER,
        permission_id INTEGER,
     PRIMARY KEY(group_id, permission_id),
         FOREIGN KEY(group_id) REFERENCES tf_group (id),
         FOREIGN KEY(permission_id) REFERENCES tf_permission (id));
```

Notice the two auxilliary tables used to provide many-to-many joins between users and groups, and between groups and users. Once we have this schema in place, a common scenario is to check whether a particular user has a particular permission. In SQL, we might write:

```
SELECT COUNT(*) FROM tf_user, tf_group, tf_permission WHERE
      tf_user.user_name='rick' AND tf_user.id=user_group.user_id
      AND user_group.group_id = group_permission.group_id
      AND group_permission.permission_id = tf_permission.id
      AND permission_name='admin';
```

In a single statement, we join the three entities—user, group, and permission—together to determine whether the user "rick" has the "admin" permission.

In the object-oriented world, we would probably model the system quite differently. We would still have users, groups, and permissions, but they would probably have an ownership relationship between them:

```
class User(object):
    groups=[]

class Group(object):
    users=[]
    permissions=[]

class Permission(object):
    groups=[]
```

Suppose we wanted to print out a summary of all of a given user's groups and permissions, something an object-oriented style would do quite well. We might write something like the following:

```
print 'Summary for %s' % user.user_name
for g in user.groups:
    print '  Member of group %s' % g.group_name
    for p in g.permissions:
        print '    ... which has permission %s' % p.permission_name
```

On the other hand, if we wanted to determine whether a user has a particular permission, we would need to do something like the following:

```
def user_has_permission(user, permission_name):
    for g in user.groups:
        for p in g.permissions:
            if p.permission_name == 'admin':
                return True
    return False
```

In this case, we needed to write a nested loop, examining every group the user is a member of to see if that group had a particular permission. SQLAlchemy lets you use object-oriented programming where appropriate (such as checking for a user's permission to do something) and relational programming where appropriate (such as printing a summary of groups and permissions). In SQLAlchemy, we could print the summary information exactly as shown, and we could detect membership in a group with a much simpler query. First, we need to create mappings between our tables and our objects, telling SQLAlchemy a little bit about the many-to-many joins:

```
mapper(User, user_table, properties=dict(
    groups=relation(Group, secondary=user_group, backref='users')))
mapper(Group, group_table, properties=dict(
    permissions=relation(Permission, secondary=group_permission,
                         backref='groups')))
mapper(Permission, permission_table)
```

Now, our model plus the magic of the SQLAlchemy ORM allows us to detect whether the given user is an administrator:

```
q = session.query(Permission)
rick_is_admin = q.count_by(permission_name='admin',
... user_name='rick')
```

SQLAlchemy was able to look at our mappers, determine how to join the tables, and use the relational model to generate a single call to the database. The SQL generated by SQLAlchemy is actually quite similar to what we would have written ourselves:

```
SELECT count(tf_permission.id)
FROM tf_permission, tf_user, group_permission, tf_group, user_group
WHERE (tf_user.user_name = ?
      AND ((tf_permission.id = group_permission.permission_id
      AND tf_group.id = group_permission.group_id)
      AND (tf_group.id = user_group.group_id
      AND tf_user.id = user_group.user_id)))
      AND (tf_permission.permission_name = ?)
```

SQLAlchemy's real power comes from its ability to bridge the object/relational divide and allow you to use whichever model is appropriate to your task at hand. Aggregation is another example of using SQLAlchemy's relational model rather than the object-oriented model. Suppose we wanted a count of how many users had each permission type. In the traditional object-oriented world, we would probably loop over each permission, then over each group, and finally count the users in the group (without forgetting to remove duplicates!). This leads to something like this:

```
for p in permissions:
    users = set()
```

```
        for g in p.groups:
            for u in g.users:
                users.add(u)
    print 'Permission %s has %d users' % (p.permission_name, len(users))
```

In SQLAlchemy, we can drop into the SQL expression language to create the following query:

```
q=select([Permission.c.permission_name,
          func.count(user_group.c.user_id)],
         and_(Permission.c.id==group_permission.c.permission_id,
              Group.c.id==group_permission.c.group_id,
              Group.c.id==user_group.c.group_id),
         group_by=[Permission.c.permission_name],
         distinct=True)
rs=q.execute()
for permission_name, num_users in q.execute():
print 'Permission %s has %d users' % (permission_name, num_users)
```

Although the query is a little longer in this case, we are doing all of the work *in the database*, allowing us to reduce the data transferred and potentially increase performance substantially due to reduced round-trips to the database. The important thing to note is that SQLAlchemy makes "simple things simple, and complex things possible."

SQLAlchemy Philosophy

SQLAlchemy was created with the goal of letting your objects be objects, and your tables be tables. The SQLAlchemy home page puts it this way:

> *SQLAlchemy Philosophy*
>
> SQL databases behave less and less like object collections the more size and performance start to matter; object collections behave less and less like tables and rows the more abstraction starts to matter. SQLAlchemy aims to accommodate both of these principles.
>
> —From *http://www.sqlalchemy.org*

Using the object mapper pattern (where plain Python objects are mapped to SQL tables via a mapper object, rather than requiring persistent objects to be derived from some `Persistable` class) achieves much of this separation of concerns. There has also been a concerted effort in SQLAlchemy development to expose the full power of SQL, should you wish to use it.

In SQLAlchemy, your objects are POPOs until you tell SQLAlchemy about them. This means that it is entirely possible to "bolt on" persistence to an existing object model by mapping the classes to tables. For instance, consider an application that uses users, groups, and permissions, as shown. You might prototype your application with the following class definitions:

```
class User(object):

    def __init__(self, user_name=None, password=None, groups=None):
        if groups is None: groups = []
```

```
        self.user_name = user_name
        self.password = password
        self._groups = groups

    def join_group(self, group):
        self._groups.append(group)

    def leave_group(self, group):
        self._groups.remove(group)

class Group(object):

def __init__(self, group_name=None, users=None, permissions=None):
        if users is None: users = []
        if permissions is None: permissions = []
        self.group_name = group_name
        self._users = users
        self._permissions = permissions

    def add_user(self, user):
        self._users.append(user)

    def del_user(self, user):
        self._users.remove(user)

    def add_permission(self, permission):
        self._permissions.append(permission)

    def del_permission(self, permission):
        self._permissions.remove(permission)

class Permission(object):

    def __init__(self, permission_name=None, groups=None):
        self.permission_name = permission_name
        self._groups = groups

    def join_group(self, group):
        self._groups.append(group)

    def leave_group(self, group):
        self._groups.remove(group)
```

Once your application moves beyond the prototype stage, you might expect to have to write code to manually load objects from the database or perhaps some other kind of persistent object store. If you are using SQLAlchemy, on the other hand, you would just define your tables:

```
user_table = Table(
    'tf_user', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_name', Unicode(16), unique=True, nullable=False),
    Column('password', Unicode(40), nullable=False))

group_table = Table(
```

```
        'tf_group', metadata,
        Column('id', Integer, primary_key=True),
        Column('group_name', Unicode(16), unique=True, nullable=False))

    permission_table = Table(
        'tf_permission', metadata,
        Column('id', Integer, primary_key=True),
        Column('permission_name', Unicode(16), unique=True,
               nullable=False))

    user_group = Table(
        'user_group',  metadata,
        Column('user_id', None, ForeignKey('tf_user.id'),
               primary_key=True),
        Column('group_id', None, ForeignKey('tf_group.id'),
               primary_key=True))

    group_permission = Table(
        'group_permission',  metadata,
        Column('group_id', None, ForeignKey('tf_group.id'),
               primary_key=True),
        Column('permission_id', None, ForeignKey('tf_permission.id'),
               primary_key=True))
```

and your mappers:

```
    mapper(User, user_table, properties=dict(
    _groups=relation(Group, secondary=user_group, backref='_users')))
    mapper(Group, group_table, properties=dict(
        _permissions=relation(Permission, secondary=group_permission,
                              backref='_groups')))
    mapper(Permission, permission_table)
```

and you're done. No modification of your objects is required—they are still simply new-style (derived from the `object` class) Python classes, and they still have whatever methods you have defined, as well as a few attributes added by SQLAlchemy (described in the sidebar "Instrumentation on Mapped Classes"). Your old methods `join_group`, `leave_group`, etc. *still work*, even without modifying the class code. This means that you can modify mapped "collection" properties (properties modeling 1:N or M:N relationships) with regular list operations, and SQLAlchemy will track your changes and flush them to the database automatically.

Instrumentation on Mapped Classes

Mapped classes are actually fairly unmolested by the default SQLAlchemy mapper. In particular, the mapped class is given the following new attributes:

`c`
: This attribute contains a collection of the columns in the table being mapped. This is useful when constructing SQL queries based on the mapped class, such as referring to `User.c.user_name`.

`_state`
: SQLAlchemy uses this property to track whether a mapped object is "clean" (freshly fetched from the databaes), "dirty" (modified since fetching from the database), or "new" (as-yet unsaved to the database). This property generally should not be modified by the application programmer.

`mapped properties`
: One attribute will be added to the mapped class for each property specified in the mapper, as well as any "auto-mapped" properties, such as columns. In the previous example, the mapper adds `user_name`, `password`, `id`, and `_groups` to the `User` class.

So, if you are planning on using SQLAlchemy, you should stay away from naming any class attributes `c` or `_state`, and you should be aware that SQLAlchemy will instrument your class based on the properties defined by the mapper.

SQLAlchemy also allows you the full expressiveness of SQL, including compound (multi-column) primary keys and foreign keys, indices, access to stored procedures, the ability to "reflect" your tables from the database into your application, and even the ability to specify cascading updates and deletes on your foreign key relationships and value constraints on your data.

SQLAlchemy Architecture

SQLALchemy consists of several components, including the aforementioned database-independent SQL expression language object-relational mapper. In order to enable these components, SQLAlchemy also provides an `Engine` class, which manages connection pools and SQL dialects, a `MetaData` class, which manages your table information, and a flexible type system for mapping SQL types to Python types.

Engine

The beginning of any SQLAlchemy application is the `Engine.` The engine manages the SQLAlchemy connection pool and the database-independent SQL dialect layer. In our previous examples, the engine was created implicitly when the `MetaData` was created:

```
metadata=MetaData('sqlite://')
engine = metadata.bind
```

It is also possible to create an engine manually, using the SQLAlchemy function `create_engine( )`:

```
engine=create_engine('sqlite://')
```

This engine can later be bound to a `MetaData` object just by setting the `bind` attribute on the `MetaData`:

```
metadata.bind = engine
```