Introduction to SQLAlchemy

Release 1

Michael Bayer

June 17, 2013

Contents:

CHAPTER

ONE

FRONT MATTER

1.1 Purpose of this Document

Students attending the *Introduction to SQLAlchemy* tutorial should receive a copy of this document prior to attending the class, and hopefully will be able to download the entire student handout package.

At the very least, students should read through the *Relational Database Review* section of this document, which introduces/reviews basic relational database and transactional concepts; this section is essentially prerequisite material to the tutorial itself. As many/most students will be familiar with a large portion of this material, making it available ahead of time will hopefully allow those who are less familiar to catch up, while not spending too much time in the class itself with what may be review material for many.

As an optional task, students may also install and test the slide runner environment and application; the *Package Setup* section describes these steps in detail, and includes getting students familiar with Python virtual environments if not already, getting the latest version of SQLAlchemy installed into the local (non-system-wide) environment, and ensuring that the tutorial slides can be run successfully. In the actual class, we'll spend a lot of time stepping through these slides, and students can step through the same slides locally on their machines as well as attempt other experiments and exercises within the environment.

Two other sections, *Glossary* and *Further Reading*, represent more detailed paths for learning. The glossary is broken into two sections: *Relational Terms* and *SQLAlchemy Core / Object Relational Terms*. It's a good idea for students to run through the relational section here; the SQLAlchemy section on the other hand will be more useful after the tutorial is complete, as a place to review some of the key concepts covered in the material.

1.2 Web Site

http://www.sqlalchemy.org/

1.3 Mailing List

Send an email to: sqlalchemy-subscribe@googlegroups.com

View archives or subscribe with a Google account at: http://groups.google.com/group/sqlalchemy

1.4 IRC Channel

#sqlalchemy on the Freenode network

1.5 Presenters / Credits

Michael Bayer (http://techspot.zzzeek.org) is a NYC-based software contractor with a decade of experience dealing with relational databases of all shapes and sizes. After writing many homegrown database abstraction layers in such languages as C, Java and Perl, and finally after several years of practice working with a huge multi-server Oracle system for Major League Baseball, he wrote SQLAlchemy as the 'ultimate toolset' for generating SQL and dealing with databases overall. The goal is to contribute towards a world-class one-of-a-kind toolset for Python, helping to make Python the universally popular programming platform it deserves to be.

Parts of this handout, as well as the core of the "Slide Presenter" tool, were written by Jason Kirtland (http://discorporate.us/jek/) who has been a key contributor to the SQLAlchemy project for many years.

Big thanks to Ben Trofatter for help editing / proofreading this document.

CHAPTER

TWO

PACKAGE SETUP

2.1 Contents

This package contains:

- student handout built as a PDF file `handout.pdf`.
- Interactive Python "slide runner" application, which is essentially a customized REPL that can step through segments of a Python script.
- Demonstration Python scripts which illustrate various features of SQLAlchemy; these scripts are formatted to work best with the "slide runner" application, though can be run directly as well.
- Packages required to run the interactive slide runner and the example SQLAlchemy programs in `sw/`, including SQLAlchemy itself, as well as virtualenv.

2.2 Obtaining the Package

A zipfile of the package is currently available on the techspot blog:

`http://techspot.zzzeek.org/sqlalchemy_tutorial.zip`

2.3 Environment Install Prerequisites

This section will discuss the prerequisites to installing the sample SQLAlchemy environment and slide runner application.

A minimum version of Python 2.6 is recommended; Python 2.7, 3.1, 3.2 or 3.3 are also fine.

For database access, the tutorials use the SQLite database by default, which is included as part of the Python standard library.

If your Python was custom built and does not include SQLite, it can be added in by rebuilding with the SQLite libraries available or by installing pysqlite.

2.4 Installing the Slide Environment

The slide environment features a working SQLAlchemy environment as well as several tutorial-style Python scripts which illustrate usage patterns. The slides are best run using a specialized "slide runner" application, which we will be running as part of the class.

To make the installation as easy as possible, as well as to minimize the need for network access, source installation packages for the non- standard prerequisite libraries are included here in the `sw/` directory.

Steps to install:

1. If virtualenv is not installed locally, or if you're not sure, the `install_venv.py` script will install using a local virtualenv script. Assuming a Python intepreter is in the path:

   ```
   $ python install_venv.py
   ```

   This script will create a virtual Python environment in the local directory `.venv` into which it will then run the `install.py` script to install the rest of the libraries.

2. If the local workstation does have virtualenv installed, it can be run manually:

   ```
   $ virtualenv .venv
   ```

   This will create a virtual Python environment in the directory `.venv`. The `install.py` script should then be run, which will then setup libraries in this environment. On Linux/OSX:

   ```
   $ .venv/bin/python install.py
   ```

   On Windows:

   ```
   $ .venv\Scripts\python.exe install.py
   ```

3. Once `.venv` is present and the libraries are installed, a particular tutorial script can be run using the `sliderepl` program.

   On Linux/OSX:

   ```
   $ .venv/bin/sliderepl 01_engine_usage.py
   ```

   On Windows:

   ```
   $ .venv\Scripts\sliderepl.exe 01_engine_usage.py
   ```

CHAPTER

THREE

RELATIONAL DATABASE REVIEW

3.1 Introduction

This document is a brief overview of key relational database concepts, including SQL basics as well as the basics of transactions. SQLAlchemy is somewhat unique in that it doesn't try to hide any of these concepts within regular use, and the developer of a SQLAlchemy-enabled application will be dealing with concepts of SQL, transactions, and Python expressions and object models, all at the same time. While this may sound daunting, it is in practice actually a better way to work, instead of relying upon a tool to hide away the existence of the relational database. Best of all, once the practice of integrating all three techniques is mastered, you'll be able to call yourself an Alchemist :).

The *Introduction to SQLAlchemy* tutorial starts off with the assumption that the student is familiar with the concepts outlined in this document - if they're new to you, spending some time familiarizing will be time well spent, and if they're old hat, you'll be in good shape to jump right into the SQLAlchemy tutorial.

Throughout this document, we'll try to refer heavily to the *Glossary*, which tries to provide an overview and additional links on just about every concept.

3.2 Overview

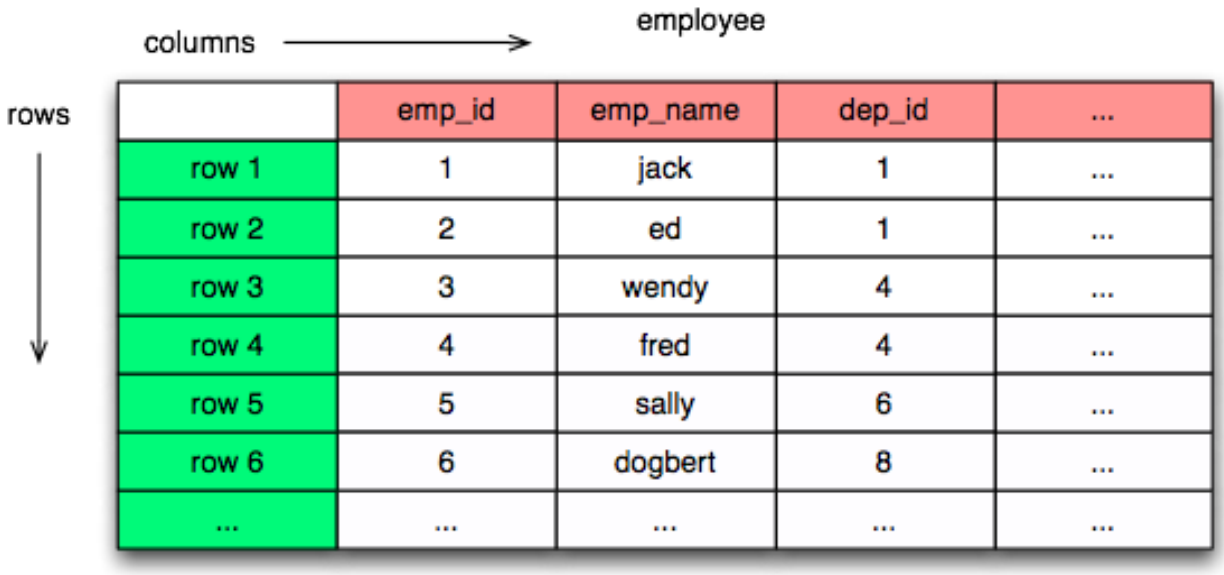
- Relational Databases, or RDBMS, are databases that draw upon the *Relational Model*, created by *Edgar Codd*.
- RDBMS organizes data into tables, rows, and columns, mimicking similar concepts in the relational model.
- The relational model encourages *normalization*, which is a system of minimizing repetition and dependency between rows.
- RDBMSs use *Structured Query Language* to access and manipulate rows.
- RDBMSs provide guarantees for data using the *ACID* model.

3.3 Relational Schemas

The *schema* refers to a fixed structure configured within a database that defines how data will be represented. The most fundamental unit of data within a schema is known as the *table*.

3.3.1 Table

The Table is the basic unit of storage in a relational database, representing a set of rows.



The table encompasses a series of *columns*, each of which describes a particular type of data unit within the table. The data within the table is then organized into *rows*, each row containing a single value for each column represented in the table.

3.3.2 DDL

At the SQL console, we create a new table as a permanent fixture within a database schema using the `CREATE TABLE` statement. The `CREATE TABLE` statement is an example of Data Definition Language, or *DDL*, which is a subset of SQL:

```
CREATE TABLE employee (
     emp_name VARCHAR(30),
     dep_id INTEGER
)
```

3.3.3 Primary Keys

A table can be created with *constraints*, which place rules on what specific data values can be present in the table. One of the most common constraints is the *primary key constraint*, which enforces that every row of the table must have a uniquely identifying value, consisting of one or more columns, where the values can additionally not be NULL. A primary key that uses more than one column to produce a value is known as a *composite* primary key.

It is a best practice that all tables in a relational database contain a primary key. Two varieties of primary key are *surrogate primary key* and *natural primary key*, where the former is specifically a "meaningless" value, and the latter is "meaningful". Which style to use is a hotly debated topic; the surrogate key is generally chosen for pragmatic reasons, including memory and index performance as well as simplicity when dealing with updates, whereas the natural primary key is often chosen for being more "correct" and closer to the relational ideal. We restate our `employee` table below adding a surrogate integer primary key on a new column `emp_id`:

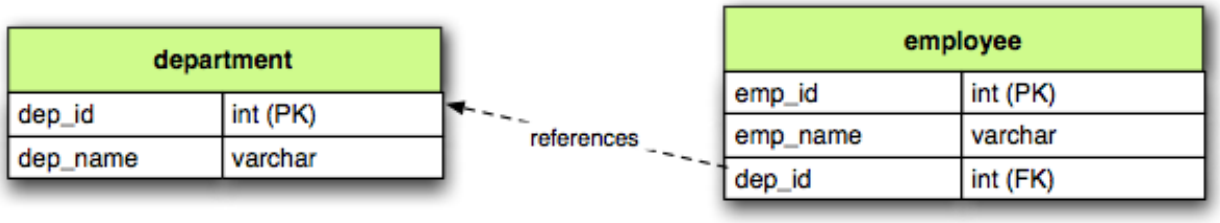
```
CREATE TABLE employee (
     emp_id INTEGER,
     emp_name VARCHAR(30),
     dep_id INTEGER,
     PRIMARY KEY (emp_id)
)
```

3.3.4 Foreign Keys

Once a table is defined as having a primary key constraint, another table can be constrained such that its rows may refer to a row that is guaranteed to be present in this table. This is implemented by establishing a column or columns in the "remote", or child, table whose values must match a value of the primary key of the "local", or parent, table. Both sets of columns are then named as members of a *foreign key constraint*, which instructs the database to enforce that values in these "remote" columns are guaranteed to be present in the "local" table's set of primary key columns. This constraint takes effect at every turn; when rows are inserted into the remote table, when rows are modified in the remote table, as well as when an attempt is made to delete or update rows in the parent table, the database ensures that any value subject to the foreign key constraint be present in the set of referenced columns, or the statement is rejected.

A foreign key constraint that refers fully to a composite primary key is predictably known as a *composite foreign key*. It is also possible, in a composite scenario, for a foreign key constraint to only refer to a subset of the primary key columns in the referenced table, but this is a highly unusual case.

Below, the figure illustrates a `department` table which is referred to by the `employee` table by relating the `employee.dep_id` column to the `department.dep_id` column:



The above schema can be created using DDL as follows:

```
CREATE TABLE department (
      dep_id INTEGER,
      dep_name VARCHAR(30),
      PRIMARY KEY (dep_id)
)

CREATE TABLE employee (
      emp_id INTEGER,
      emp_name VARCHAR(30),
      dep_id INTEGER,
      PRIMARY KEY (emp_id),
      FOREIGN KEY (dep_id)
        REFERENCES department(dep_id)
)
```

3.3.5 Normalization

The structure of a relational schema is based on a system known as *relational algebra*. The central philosophy that drives the design of a relational schema is a process known as *normalization*, which like most fundamental computer science concepts is an entire field of study onto itself. In practice however, normalization usually boils down to a few simple practices that become second nature in not too much time.

The general idea of normalization is to eliminate the repetition of data, so that any one particular piece of information is represented in exactly one place. By doing so, that piece of information becomes one of many atomic units by which data can be searched and operated upon. For example, if hundreds of records all refer to a particular date record, we can correlate all those records on this single date record strictly based on the association of those identities.

A typical example of denormalized data looks like:

```
Employee Language
------------------
name        language   department
-------     --------   -------------
Dilbert     C++        Systems
Dilbert     Java       Systems
Wally       Python     Engineering
Wendy       Scala      Engineering
Wendy       Java       Engineering
```

The table's rows can be uniquely identified by the composite of the "name" and "language" columns, which therefore make up the table's *candidate key*. Normalization theory would claim the above table violates "second normal form" because the "non prime" attribute "department" is logically dependent only on the "name" column, which is a subset of the candidate key. (Note that the author is carefully parsing the Wikipedia page for normalization here in order to state this correctly). A proper normalization would use two tables along the lines of the following:

```
Employee Department
-------------------
name        department
--------    -----------
Dilbert     Systems
Wally       Engineering
Wendy       Engineering

Employee Language
------------------
name        language
--------    --------
Dilbert     C++
Dilbert     Java
Wally       Python
Wendy       Scala
Wendy       Java
```

While the formal reasoning behind the above change may be difficult to parse, a visual inspection of the data reveals more obviously how the second form is an improvement; the original version repeats duplicate associations between "name" and "department" many times according to how many distinct "language" values correspond to a name; whereas the second version uses separate tables so that each "name/department" and "name/language" association can be expressed independently.

The concept of data constraints, particularly the primary key constraint and the foreign key constraint, are designed to work naturally with the concept of normalization. Constraints would be applied to the above schema by establishing "Employee Department->name" as a primary key, establishing "Employee Language->name, language" as a composite primary key, and then creating a foreign key such that "Employee Language->name" must refer to "Employee Department->name". When a schema resists being organized into simple primary and foreign key relationships, that's often a sign that it isn't strongly normalized.

The Wikipedia page on normalization (http://en.wikipedia.org/wiki/Database_normalization) is a great place to learn more.

3.4 Data Manipulation Language (DML)

Once we have a schema defined, data can be placed into the tables and also modified using another subset of SQL called *data manipulation language*, or DML.

3.4.1 Inserts

New rows are added to a table using the `INSERT` statement. The `INSERT` statement contains a `VALUES` clause which refers to the actual values to be inserted into each row.

```
INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (1, 'dilbert', 1);

INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (2, 'wally', 1);
```

Auto Incrementing Integer Keys

Most modern databases feature a built-in system of generating incrementing integer values, which are in particular usually used for tables that have surrogate integer primary keys, such as our `employee` and `department` tables. For example, when using SQLite, the above `emp_id` column will generate an integer value automatically; when using MySQL, an integer primary key declared with `AUTO INCREMENT` will do so as well; and on Postgresql, declaring a primary key with the datatype `SERIAL` will have the same end effect. When using these so-called "auto incrementing" primary key generators, we *omit* the column from the INSERT statement:

```
INSERT INTO employee (emp_name, dep_id)
            VALUES ('dilbert', 1);

INSERT INTO employee (emp_name, dep_id)
            VALUES ('wally', 1);
```

Databases that feature primary key generation systems will also feature some means of acquiring the "generated" integer identifier after the fact, using non-standard SQL extensions and/or functions. When using Postgresql, one such way of reading these generated identifiers is to use `RETURNING`:

```
INSERT INTO employee (emp_name, dep_id)
            VALUES ('dilbert', 1) RETURNING emp_id;

emp_id
------
  1
```

While every database features a different system of generating and retrieving these keys, we'll generally refer to the style above where the integer primary key can be omitted from an INSERT. When using SQLAlchemy, one of the most fundamental features it provides is a consistent and transparent system of utilizing the wide variety of key generation and retrieval schemes.

3.4.2 Updates

The `UPDATE` statement changes the contents of an existing row, using a `WHERE` clause to identify those rows which are the target of the update, and a `SET` clause which identifies those columns which should be modified and to what values:

```
UPDATE employee SET dep_id=7 WHERE emp_name='dilbert'
```

When an UPDATE statement like the above one executes, it may match any number of rows, including none at all. An UPDATE statement typically has a "row count" value associated with a particular execution, which indicates the number of rows that matched the WHERE criteria, and therefore represents the number of rows that were subject to the SET clause.

3.4.3 Deletes

The `DELETE` statement removes rows. Like the `UPDATE` statement, it also uses a `WHERE` clause to identify those rows which should be deleted:

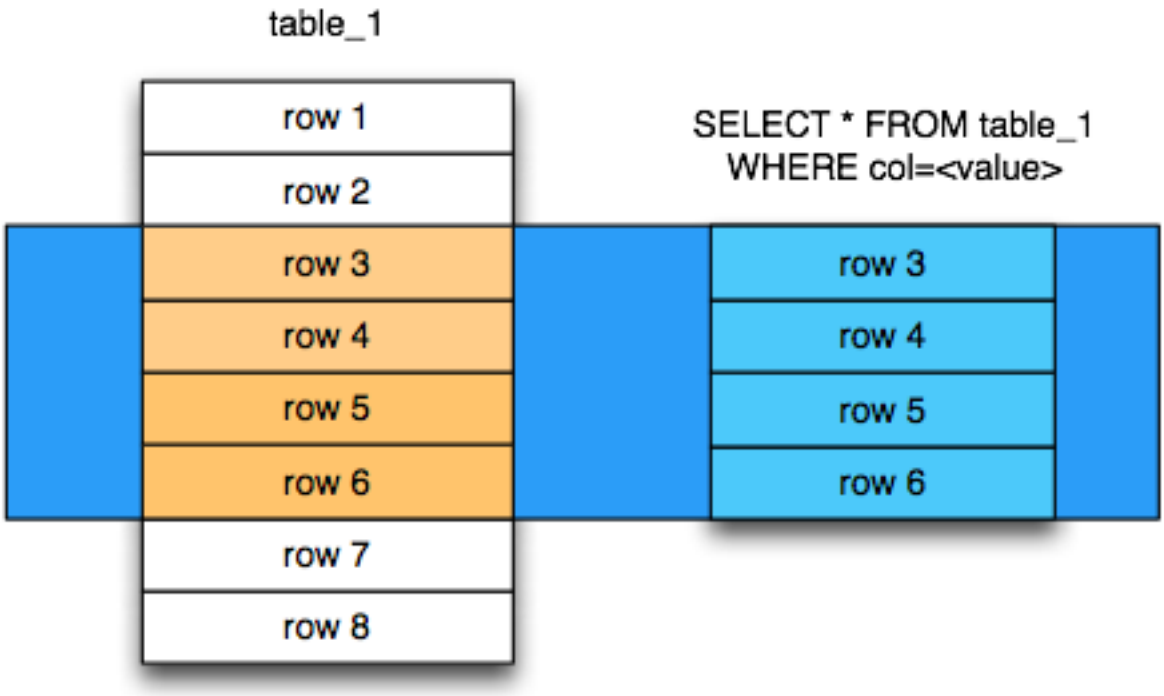
```
DELETE FROM employee WHERE dep_id=1
```

Above, all employee records within department id 1 will be deleted.

3.5 Queries

The key feature of SQL is its ability to issue queries. The `SELECT` statement is the primary language construct providing this feature, and is where we spend most of our time when using relational databases, allowing us to query for rows in tables.

An illustration of a `SELECT` statement is in the figure below. Like the `UPDATE` and `DELETE` statements, it also features a `WHERE` clause which is the primary means of specifying which rows should be selected.



An example of a `SELECT` that chooses the rows where `dep_id` is equal to the value `12`:

```
SELECT emp_id, emp_name FROM employee WHERE dep_id=12
```

The key elements of the above `SELECT` statement are:

1. The *FROM clause* determines the table or tables from which we are to select rows.
2. The *WHERE clause* illustrates a criterion which we use to filter those rows retrieved from the tables in the `FROM` clause
3. The *columns clause* is the list of expressions following the `SELECT` keyword and preceding the `FROM` keyword, and indicates those values which we'd like to display given each row that we've selected.

With the above rules, our statement might return to us a series of rows that look like this, if the `emp_name` column values `wally`, `dilbert`, and `wendy` were all those linked to `dep_id=12`:

```
emp_id    emp_name
-------+------------
  1    |   wally
  2    |   dilbert
  5    |   wendy
```

3.5.1 Ordering

The `ORDER BY` clause may be applied to a `SELECT` statement to determine the order in which rows are returned. Ordering is applied to the `SELECT` after the `WHERE` clause. Below, we illustrate our statement loading employee records ordered by name:

```
SELECT emp_id, emp_name FROM employee WHERE dep_id=12 ORDER BY emp_name
```

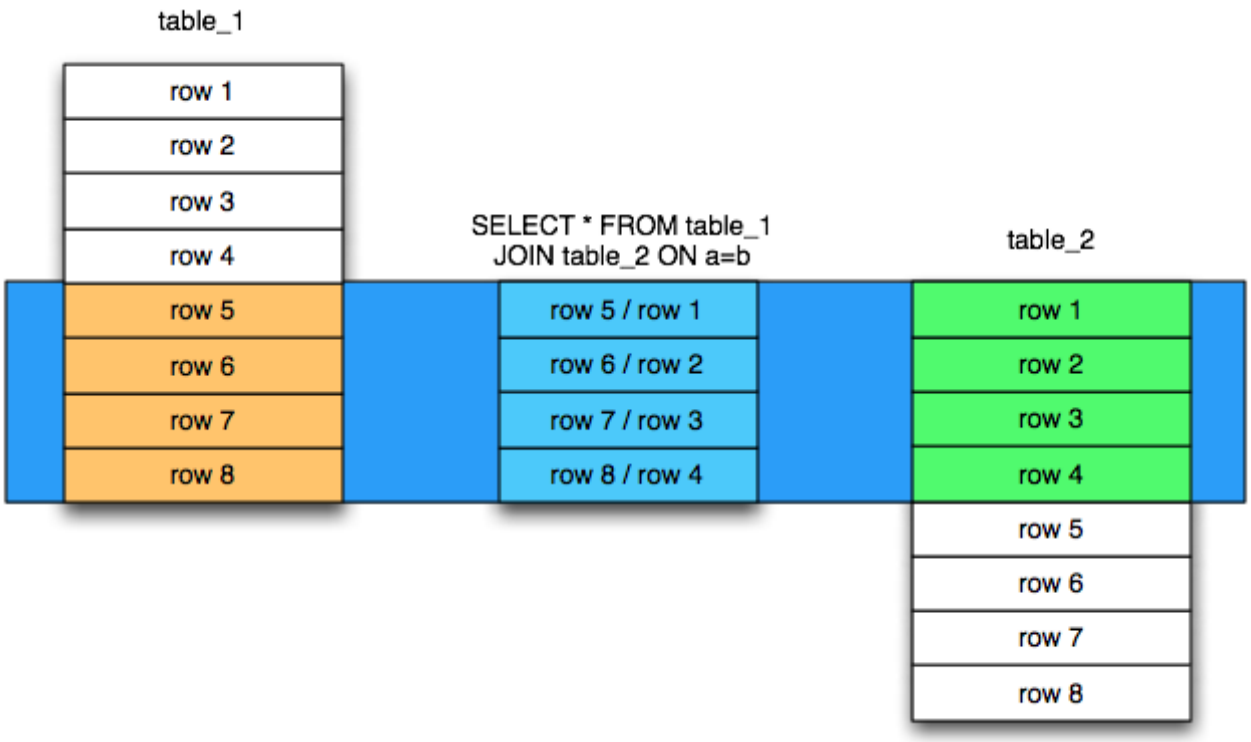
Our result set then comes back like this:

```
emp_id    emp_name
-------+------------
  2    |   dilbert
  1    |   wally
  5    |   wendy
```

3.5.2 Joins

A `SELECT` statement can use a *join* to produce rows from two tables at once, usually joining along foreign key references. The `JOIN` keyword is used in between two table names inside the `FROM` clause of a `SELECT` statement. The `JOIN` also usually includes an `ON` clause, which specifies criteria by which the rows from both tables are correlated to each other.

The figure below illustrates the behavior of a join, by indicating in the central blue box those rows which are *composites* of rows from both "table_1" and "table_2" and which satisfy the `ON` clause:



It's no accident that the blue box looks a lot like a table. Even though above, only "table_1" and "table_2" represent fixed tables, the `JOIN` creates for us what is essentially a *derived table*, a list of rows that we could use in subsequent expressions.

Using our department / employee example, to select employees along with their department name looks like:

```
SELECT e.emp_id, e.emp_name, d.dep_name
    FROM employee AS e
    JOIN department AS d
      ON e.dep_id=d.dep_id
   WHERE d.dep_name = 'Software Artistry'
```

The result set from the above might look like:

```
emp_id    emp_name         dep_name
-------+------------+--------------------
  2    |   dilbert  |  Software Artistry
  1    |   wally    |  Software Artistry
  5    |   wendy    |  Software Artistry
```

3.5.3 Left Outer Join

A variant of the join is the *left outer join*. This structure allows rows to be returned from the table on the "left" side which don't have any corresponding rows on the "right" side. For instance, if we wanted to select departments and their employees, but we also wanted to see the names of departments that had no employees, we might use a `LEFT OUTER JOIN`:

```
SELECT d.dep_name, e.emp_name
    FROM department AS d
    LEFT OUTER JOIN employee AS e
    ON d.dep_id=e.dep_id
```

Supposing our company had three departments, where the "Sales" department was currently without any employees, we might see a result like this:

```
     dep_name           emp_name
--------------------+-------------
 Management         |   dogbert
 Management         |   boss
 Software Artistry  |   dilbert
 Software Artistry  |   wally
 Software Artistry  |   wendy
 Sales              |   <NULL>
```

There is also a "right outer join", which is the same as left outer join except you get all rows on the right side. However, the "right outer join" is not commonly used, as the "left outer join" is widely accepted as proper convention, and is arguably less confusing than a right outer join (in any case, right outer joins confuse the author!).

3.5.4 Aggregates

An *aggregate* is a function that produces a single value, given many values as input. A commonly used aggregate function is the `count()` function which, given a series of rows as input, returns the count of those rows as an integral value. The `count()` function accepts as an argument any SQL expression, which we often pass as the wildcard string `*` that essentially means "all columns" - unlike most aggregate functions, `count()` doesn't evaluate the meaning its argument, it only counts how many times it is called:

```
SELECT count(*) FROM employee

?count?
-------

   18
```

Another aggregate expression might return to us the average number of employees within departments. To accomplish this, we also make use of the `GROUP BY` clause, described below, as well as a *subquery*:

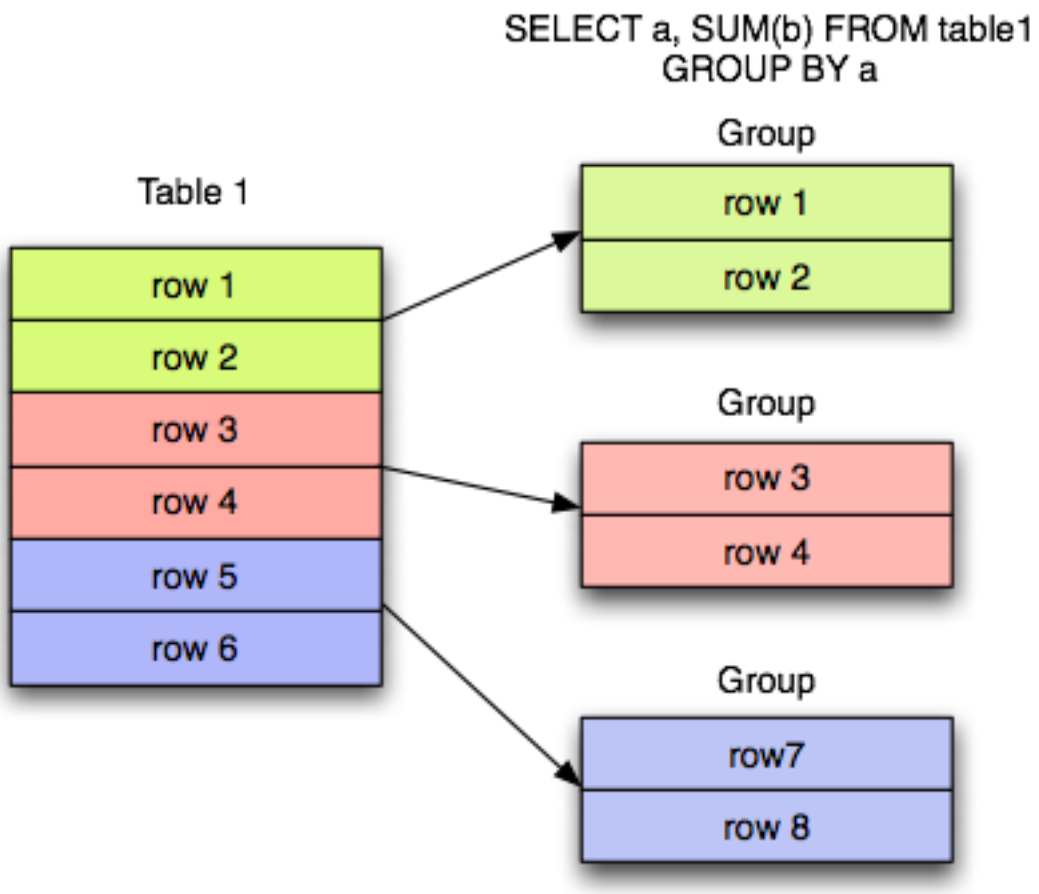
```
SELECT avg(emp_count) FROM
  (SELECT count(*) AS emp_count
    FROM employee GROUP BY dep_id) AS emp_counts

?avg?
-----
  2
```

Note the above query only takes into account non-empty departments. To include empty departments would require a more complex sub-query that takes into account rows from `department` as well.

3.5.5 Grouping

The `GROUP BY` keyword is applied to a `SELECT` statement, breaking up the rows it selects into smaller sets based on some criteria. `GROUP BY` is commonly used in conjunction with aggregates, as it can apply individual subsets of rows to the aggregate function, yielding an aggregated return value for each group. The figure below illustrates the rows from a table being broken into three sub-groups, based on the expression "a", and then the `SUM()` aggregate function applied to the value of "b" for each group:



An example of an aggregation / `GROUP BY` combination that gives us the count of employees per department id:

```
SELECT count(*) FROM employee GROUP BY dep_id
```

The above statement might give us output such as:

```
?count?  |   dep_id
---------+----------
    2    |      1
    10   |      2
    6    |      3
    9    |      4
```

3.5.6 Having

After we've grouped things with `GROUP BY` and gotten aggregated values by applying aggregate functions, we can be filter those results using the `HAVING` keyword. We can take the above result set and return only those rows where more than seven employees are present:

```
SELECT count(*) as emp_count FROM employee GROUP BY dep_id HAVING emp_count > 7
```

The result would be:

```
emp_count  |   dep_id
-----------+----------
```

```
    10      |      2
     9      |      4
```

3.5.7 SELECT Process Summary

It's very helpful (at least the author thinks so) to keep straight exactly how `SELECT` goes about its work when given a combination of aggregation and clauses (such as `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`).

Given a series of rows:

```
emp_id    emp_name     dep_id
-------+------------+----------
  1    |   wally    |      1
  2    |   dilbert  |      1
  3    |   jack     |      2
  4    |   ed       |      3
  5    |   wendy    |      1
  6    |   dogbert  |      4
  7    |   boss     |      3
```

We'll analyze what a `SELECT` statement like the following does in a logical sense:

```
SELECT count(emp_id) as emp_count, dep_id
    FROM employee
    WHERE dep_id=1 OR dep_id=3 OR dep_id=4
    GROUP BY dep_id
    HAVING emp_count > 1
    ORDER BY emp_count, dep_id
```

1. the `FROM` clause is operated upon first. The table or tables which the statement is to retrieve rows from are resolved; in this case, we start with the set of all rows contained in the `employee` table:

   ```
   ... FROM employee ...

   emp_id    emp_name     dep_id
   -------+------------+----------
     1    |   wally    |      1
     2    |   dilbert  |      1
     3    |   jack     |      2
     4    |   ed       |      3
     5    |   wendy    |      1
     6    |   dogbert  |      4
     7    |   boss     |      3
   ```

2. For the set of all rows in the `employee` table, each row is tested against the criteria specified in the `WHERE` clause. Only those rows which evaluate to "true" based on this expression are returned. We now have a subset of rows retrieved from the `employee` table:

   ```
   ... WHERE dep_id=1 OR dep_id=3 OR dep_id=4 ...

   emp_id    emp_name     dep_id
   -------+------------+----------
     1    |   wally    |      1
     2    |   dilbert  |      1
     4    |   ed       |      3
     5    |   wendy    |      1
     6    |   dogbert  |      4
     7    |   boss     |      3
   ```

3. With the target set of rows assembled, `GROUP BY` then organizes the rows into groups based on the criterion given. The "intermediary" results of this grouping will be passed on to the next step behind the scenes. Were we able to look into the pipeline, we'd see something like this:

```
... GROUP BY dep_id ...
```

```
 "group"    emp_id    emp_name     dep_id
----------+---------+------------+---------
dep_id=1   |    1    |   wally    |    1
           |    2    |   dilbert  |    1
           |    5    |   wendy    |    1
----------+---------+------------+---------
dep_id=3   |    4    |   ed       |    3
           |    7    |   boss     |    3
----------+---------+------------+---------
dep_id=4   |    6    |   dogbert  |    4
```

4. Aggregate functions are now applied to each group. We've passed emp_id to the `count()` function, which means for group "1" it will receive the values "1", "2", and "5", for group "3" it will receive the values "4" and "7", for group "4" it receives the value "6". `count()` doesn't actually care what the values are, and we could as easily have passed in `*`, which means "all columns". However, most aggregate functions do care what the values are, including functions like `max()`, `avg()` `min()` etc., so it's usually a good habit to be aware of the column expression here. Below, we observe that the "emp_id" and "emp_name" columns go away, as we've aggregated on the count:

```
... count(emp_id) AS emp_count ...
```

```
  emp_count     dep_id
------------+-----------
     3      |    1
------------+-----------
     2      |    3
------------+-----------
     1      |    4
```

5. Almost through all of our keywords, `HAVING` takes effect once we have the aggregations, and acts like a `WHERE` clause for aggregate values. In our statement, it filters out groups that have one or fewer members:

```
... HAVING emp_count > 1 ...
```

```
  emp_count     dep_id
------------+-----------
     3      |    1
------------+-----------
     2      |    3
```

6. Finally, `ORDER BY` is applied. It's important to remember in SQL that relational algebra is a language of *sets*, which are inherently un-ordered. In the typical case, all of the work of selecting, aggregating, and filtering our data are done before any ordering is applied, and only right before the final results are returned to us are they ordered:

```
... ORDER BY emp_count, dep_id
```

```
  emp_count     dep_id
------------+-----------
     2      |    3
```

```
------------+-----------
     3      |     1
```

3.6 ACID Model

The flip side to the relational model employed by relational databases is the so called *transactional* model most of them provide. The acronym *ACID* refers to the principal properties of relational database transactions (as well as transactions for any kind of hypothetical database).

3.6.1 Atomicity

Atomicity allows multiple statements to proceed within a particular demarcation known as a *transaction*, which has a single point of completion known as a *commit*. A transaction is committed once all the operations within it have completed successfully. If any of the operations fail, the transaction can instead be reverted using a *rollback*, which reverses all the steps that have proceeded within the transaction, leaving the state of the database unchanged relative to before the transaction began. Atomicity refers to the fact that all of these steps proceed or fail as a single unit; it's not possible for some of the steps to succeed without all of them succeeding.

3.6.2 Consistency

Consistency encompasses the ability of the database to ensure that it always remains in a valid state after a transaction completes successfully. When we say a database is "consistent", this includes among other things that *constraints* are satisified. Some related concepts are *cascades*, and *triggers*.

Data Constraints

While consistency really refers to a complex system employed by the database to maintain data integrity, as end users we will often think of "consistent" in terms of the contraints we've established. Data constraints are programmer-established rules that are checked against changes in data as those changes are invoked against the database. Typical constraints include:

- NOT NULL constraint - value in a column may never be NULL or non-present.
- *primary key constraint* - each row must contain a single- or multi-column value that is unique across all other rows in the table, and is the single value that logically identifies the information stored in that row.
- *foreign key constraint* - a particular column or columns must contain a value that exists elsewhere in a different row, usually of a different table. The foreign key constraint is the building block by which the rows of many flat tables can be composed together to form more intricate geometries.
- *unique constraint* - similar to the primary key constraint, the unique constraint identifies any arbitrary column or set of columns that also must be unique throughout the whole table, without themselves comprising the primary key.
- *check constraint* - any arbitrary expression can be applied to a row, which will result in that row being rejected if the expression does not evaluate to "true".

Constraints are a sometimes misunderstood concept that when properly used can give a developer a strong "peace of mind", knowing that even in the face of errors, mistakes, or omissions within applications that communicate with the database, the database itself will remain in a *consistent* state, rather than running the risk of accumulating ongoing data errors that are only detected much later when it's too late. This "peace of mind" allows us to write and test our applications more quickly and boldly than we would be able to otherwise; more quickly because the relational

database already does lots of the integrity checking we'd otherwise have to write by hand, and more boldly because there is much less chance that code errors can result in corruption of data as if we hadn't used constraints.

3.6.3 Isolation

Isolation is a complex subject which in a general sense refers to the interactivity between *concurrent* transactions, that is, more than one transaction occuring at the same time. It is focused on the degree to which the work being performed by a particular transaction may be viewed by other transactions going on at the same time. The isolation of concurrent transactions is an important area of consideration when constructing an application, as in many cases the decisions that are made within the scope of a transaction may be affected by this cross-transaction visibility; the isolation behavior can also have a significant impact on database performance. While there are techniques by which one doesn't have to worry too often about isolation, in many cases dealing with the specifics of isolation is unavoidable, and no one isolation behavior is appropriate in all cases.

In practice, the level of isolation between transactions is usually placed into four general categories (there are actually a lot more categories for people who are really into this stuff):

- Read uncommitted - This is the lowest level of isolation. In this mode, transactions are subject to so-called *dirty reads*, whereby the work that proceeds within a transaction is plainly visible to other transactions as it proceeds. With dirty reads, a transaction might UPDATE a row with updated data, and this updated row is now globally visible by other transactions. If the transaction is rolled back, all the other transactions will be exposed to this rollback as well.
- Read committed - In read committed, we're no longer subject to dirty reads, and any data that we read from concurrent transactions is guaranteed to have been committed. However, as we proceed within our own transaction, we can still see the values of rows and SELECT statements change, as concurrent transactions continue to commit modifications to rows that we're also looking at.
- Repeatable Read - The next level operates at the row level, and adds the behavior such that any individual row that we read using a SELECT statement will remain consistent from that point on, relative to our transaction. That is, if we read the row with primary key "5" from the `employee` table, and in the course of our work a concurrent transaction updates the `emp_name` column from "Ed" to "Edward", when we re-SELECT this row, we will still see the value "Ed" - that is, the value of this row remains consistent from the first time we read it forward. If we ourselves attempt to update the row again, we may be subject to a conflict when we attempt to commit the transaction.

  Within repeatable read, we are still subject to the concept of a so-called *phantom read* - this refers to a row that we see in one SELECT statement that we later (or previously) do not see in a different SELECT statement, because a concurrent transaction has deleted or inserted that row since we last selected with a given set of criteria.
- Serializable - Generally considered to be the highest level of isolation, the rough idea of serializable isolation is that we no longer have phantom reads - if we select a series of N rows using a SELECT statement, we can be guaranteed that we will always get those same N rows when emitting a subsequent SELECT of the same criteria, even if concurrent transactions have INSERTed or DELETed rows from that table.

The impact of using a higher level of isolation depends much on the specifics of the database in use. Generally, lower levels of isolation are associated with higher performance and a reduced chance of *deadlocks*. Historically, this is due to the fact that a lower level of isolation has less of a need to synchronize concurrent operations using locks. However, most modern relational databases employ a concept known as *multi version concurrency control* in order to reduce or eliminate the need for locking, which is able to maintain multiple "versions" of data at the same time, each of which can be accessed by independent transactions. As a transaction commits its data, the version of data which it works with becomes the official "data of record" for the database as a whole. An MVCC scheme may still introduce performance overhead with higher isolation levels, as such systems must monitor and report so-called *serialization failures*, which are the rejection of transactions that conflict with another one executing concurrently.

3.6.4 Durability

Durability basically means that relational databases provide a guarantee that once a transaction COMMIT has succeeded, the transaction logs have been written to non-volatile storage, so the chance of the transaction's committed state being lost due to a system failure is extremely low. Durability tends to be something most developers take for granted when working with relational databases; however, in recent years it's been discussed a lot more with the rise of so-called NoSQL databases, which in some cases attempt to scale back the promise of durability in exchange for faster transaction throughput.

CHAPTER

FOUR

GLOSSARY

The glossary is broken into two distinct areas of terminology, for those who want to read the whole thing.

Relational Terms

SQLAlchemy Core / Object Relational Terms

4.1 Relational Terms

ACID, ACID model An acronym for "Atomicity, Consistency, Isolation, Durability"; a set of properties that guarantee that database transactions are processed reliably. (via Wikipedia)

See also:

ACID Model

http://en.wikipedia.org/wiki/ACID_Model

atomicity Atomicity is one of the components of the *ACID* model, and requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. (via Wikipedia)

See also:

Atomicity

http://en.wikipedia.org/wiki/Atomicity_(database_systems)

candidate key A *relational algebra* term referring to an attribute or set of attributes that form a uniquely identifying key for a row. A row may have more than one candidate key, each of which is suitable for use as the primary key of that row. The primary key of a table is always a candidate key.

See also:

Primary Keys

http://en.wikipedia.org/wiki/Candidate_key

cartesian product A mathematical operation which returns a set (or product set) from multiple sets. The Cartesian product is the result of crossing members of each set with one another. (via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Cartesian_product

check constraint A check constraint is a condition that defines valid data when adding or updating an entry in a table of a relational database. A check constraint is applied to each row in the table.

(via Wikipedia)

A check constraint can be added to a table in standard SQL using *DDL* like the following:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

See also:

http://en.wikipedia.org/wiki/Check_constraint

column, columns A vertical unit of storage in a *table*. The table defines one or more columns as fixed types of data to be stored within rows.

columns clause The portion of a `SELECT` statement that enumerates a series of SQL expressions to be evaulated as the returned result set. Typically, these expressions refer directly to table columns. The columns clause follows the `SELECT` keyword and precedes the `FROM` keyword.

In the following `SELECT` statement, the "id" and "name" columns will be returned for each row, and this enumeration of columns forms the "columns clause":

```
SELECT id, name FROM user_account
```

commit Denotes the successful completion of a *transaction*. In SQL, we normally denote the commit using the `COMMIT` statement:

```
BEGIN TRANSACTION

INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (1, 'dilbert', 1);

INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (2, 'wally', 1);

COMMIT
```

Above, the `employee` rows for `dilbert` and `wally` will be permanently available following the `COMMIT` statement.

consistency Consistency is one of the compoments of the *ACID* model, and ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to *constraints*, cascades, triggers, and any combination thereof. (via Wikipedia)

See also:

Consistency

http://en.wikipedia.org/wiki/Consistency_(database_systems)

constraint, constraints Rules established within a relational database that ensure the validity and consistency of data. Common forms of constraint include *primary key constraint*, *foreign key constraint*, and *check constraint*.

See also:

Consistency

correlated subquery, correlated subqueries A *subquery* is correlated if it depends on data in the enclosing `SELECT`.

Below, a subquery selects the aggregate value `MIN(a.id)` from the `email_address` table, such that it will be invoked for each value of `user_account.id`, correlating the value of this column against the `email_address.user_account_id` column:

```
SELECT user_account.name, email_address.email
 FROM user_account
 JOIN email_address ON user_account.id=email_address.user_account_id
 WHERE email_address.id = (
    SELECT MIN(a.id) FROM email_address AS a
    WHERE a.user_account_id=user_account.id
 )
```

The above subquery refers to the `user_account` table, which is not itself in the `FROM` clause of this nested query. Instead, the `user_account` table is recieved from the enclosing query, where each row selected from `user_account` results in a distinct execution of the subquery.

A correlated subquery is nearly always present in the *WHERE clause* or *columns clause* of the enclosing `SELECT` statement, and never in the *FROM clause*; this is because the correlation can only proceed once the original source rows from the enclosing statement's FROM clause are available.

data definition language, DDL The SQL commands that define a schema. For example, `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`.

See also:

Relational Schemas

http://en.wikipedia.org/wiki/Data_Definition_Language

data manipulation language, DML The SQL commands that manipulate data. For example, `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

See also:

Data Manipulation Language (DML)

http://en.wikipedia.org/wiki/Data_Manipulation_Language

durability Durability is a property of the *ACID* model which means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). (via Wikipedia)

See also:

Durability

http://en.wikipedia.org/wiki/Durability_(database_systems)

Edgar Codd, Edgar F. Codd Creator of the *relational model*.

See also:

http://en.wikipedia.org/wiki/Edgar_F._Codd

EXISTS, EXISTS operator The EXISTS operator tests a subquery and returns true if the subquery returns any rows:

```
SELECT name FROM user_account
 WHERE EXISTS
 (SELECT * FROM email_address
    WHERE email_address.user_account_id=user_account.id)

 name
-------
 jack
 ed
 wendy
```

The columns selected by the subquery are ignored. Only the number of rows are considered: no rows or at least one. `EXISTS <subquery>` is a *scalar*, boolean expresion and can be used like any other boolean value in a WHERE clause:

```
SELECT name FROM user_account
  WHERE EXISTS (SELECT * FROM email_address WHERE email_address.user_account_id=user_account.id)
  AND name='ed'

 name
------
  ed
```

The subquery used within an `EXISTS` expression is nearly always a *correlated subquery*.

foreign key constraint A referential constraint between two tables. A foreign key is a field or set of fields in a relational table that matches a *candidate key* of another table. The foreign key can be used to cross-reference tables. (via Wikipedia)

A foreign key constraint can be added to a table in standard SQL using *DDL* like the following:

```
ALTER TABLE employee ADD CONSTRAINT dep_id_fk
FOREIGN KEY (employee) REFERENCES department (dep_id)
```

See also:

Foreign Keys

http://en.wikipedia.org/wiki/Foreign_key_constraint

FROM clause A component of the `SELECT` statement which specifies the source tables or subqueries from which rows are to be selected. The `FROM` clause follows the *columns clause* and may contain a comma-separated list of tables and subqueries, as well as *join* expressions:

```
-- FROM clause illustrating an explicit join

SELECT id, name, email_address
 FROM user_account
 JOIN email_address ON user_account.id=email_address.user_account_id

-- FROM clause illustrating an implicit join

SELECT id, name, email_address
 FROM user_account, email_address
 WHERE user_account.id=email_address.user_account_id
```

IN, IN operator A comparison operator. Compares an expression against a list of values, and is true if it matches at least one of them.

```
SELECT email FROM email_address
WHERE user_account_id IN (1, 2)
```

A *subquery* can be used in place of a literal list of values:

```
SELECT email FROM email_address
WHERE user_account_id IN
(SELECT id FROM user_account WHERE name='jack' OR name='ed')
```

isolation, isolated The isolation property of the *ACID* model ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Each transaction must execute in total isolation i.e. if T1 and T2 execute concurrently then each should remain independent of the other.[citation needed] (via Wikipedia)

See also:

Isolation

http://en.wikipedia.org/wiki/Isolation_(database_systems)

join, inner join Combines the rows of two tables. Considers each pair of rows in turn, and returns one combined row for each pair that matches an ON criteria.

```
SELECT ua.id, ua.name, ea.email, ea.user_account_id
 FROM user_account AS ua
  JOIN email_address AS ea
  ON ua.id = ea.user_account_id

 id | name  |      email      | user_account_id
----+-------+-----------------+----------------
  1 | jack  |  jack@jack.com  |       1
  2 | ed    |  ed@yahoo.com   |       2
  2 | ed    |  ed@msn.com     |       2
  3 | wendy |  wendy@nyt.com  |       3
```

The result of the join can be defined in a logical sense by first determining the *cartesian product* of the left and right side tables; then, for each row within this product, evaluating `ON` clause for each row, selecting only those rows for which the clause evaluates to "true". In practice, relational database systems use more efficient approaches internally in order to evaluate the result of a join.

Usage of the `JOIN` or `INNER JOIN` keyword is logically equivalent to a so-called *implicit join*, where the `JOIN` keyword is not present, and instead the left and right side expressions are delivered to the *FROM clause* as a comma separated list, with the ON criteria stated instead in the `WHERE` clause:

```
SELECT ua.id, ua.name, ea.email, ea.user_account_id
 FROM user_account AS ua, email_address.ea
 WHERE ua.id = ea.user_account_id
```

See also:

left outer join

http://en.wikipedia.org/wiki/Sql_join

left outer join A variant of the *join* whereby the criteria for including rows from the "left" side is relaxed, such that not only left-side rows which correspond to the right side are returned, but also left-side rows for which no right side row corresponds. In the case where no right side row corresponds, all columns from the right side are returned as NULL.

Below, we illustrate selecting all user names from the `user_account` table, in addition to all the `email_address` rows for each `user_account` row, but also including rows from `user_account` for which no row in `email_address` is present:

```
SELECT ua.id, ua.name, ea.email, ea.user_account_id
 FROM user_account AS ua
  JOIN email_address AS ea
  ON ua.id = ea.user_account_id

 id | name  |      email      | user_account_id
----+-------+-----------------+----------------
  1 | jack  |  jack@jack.com  |       1
  2 | ed    |  ed@yahoo.com   |       2
  2 | ed    |  ed@msn.com     |       2
  3 | wendy |  wendy@nyt.com  |       3
  4 | mary  |     (null)      |     (null)
```

The left outer join is a key technique used in object relational systems in order to resolve a *one to many* collection, that is a series of objects that contain zero or more related objects.

See also:

join

multi version concurrency control, MVCC A system by which modern databases provide concurrent access to database data. By assigning *versions* to snapshots of data in time, multiple transactions may simultaneously view different versions of the data, relative to the time that they were begun.

See also:

Isolation

http://en.wikipedia.org/wiki/Multiversion_concurrency_control

natural primary key A *primary key* that is formed of attributes that already exist in the real world. For example, a USA citizen's social security number could be used as a natural key. In other words, a natural key is a *candidate key* that has a logical relationship to the attributes within that *row*.

(via Wikipedia)

See also:

surrogate primary key

http://en.wikipedia.org/wiki/Natural_key

normalization Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency. Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships. (via Wikipedia)

See also:

Normalization

http://en.wikipedia.org/wiki/Database_normalization

primary key, primary key constraint A *constraint* that uniquely defines the characteristics of each *row*. The primary key has to consist of characteristics that cannot be duplicated by any other row. The primary key may consist of a single attribute or multiple attributes in combination. (via Wikipedia)

The primary key of a table is typically, though not always, defined within the `CREATE TABLE` *DDL*:

```
CREATE TABLE employee (
     emp_id INTEGER,
     emp_name VARCHAR(30),
     dep_id INTEGER,
     PRIMARY KEY (emp_id)
)
```

See also:

Primary Keys

http://en.wikipedia.org/wiki/Primary_Key

query, queries The means of interrogating a relational database for data. The primary feature in SQL used for querying is the `SELECT` statement.

See also:

Queries

http://en.wikipedia.org/wiki/Sql#Queries

relation, relations In *relational algebra*, a single grid of data represented by zero or more *tuples*. In a SQL database, the most common relation is the *table*, which defines one or more columns of zero or more *rows*. The output of a `SELECT` statement is also a relation.

relational model, relational algebra The relational model for database management is a database model based on first-order predicate logic, first formulated and proposed in 1969 by *Edgar F. Codd*. In the relational model of a database, all data is represented in terms of *tuples*, grouped into *relations*. A database organized in terms of the relational model is a relational database. (via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Relational_model

right outer join Like a *left outer join*, except the left and right side are swapped. At least one row will be returned for every row in the right table, and columns from the left row will be filled with NULL if the ON criteria does not match. Right outer joins are not frequently used.

rollback Denotes the end to a *transaction* which reverses all the effects of the transaction that have proceeded thus far; the state established within the transaction is discarded. In SQL, this is normally denoted using the `ROLLBACK` statement:

```
BEGIN TRANSACTION

INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (1, 'dilbert', 1);

INSERT INTO employee (emp_id, emp_name, dep_id)
            VALUES (2, 'wally', 1);

ROLLBACK
```

Above, no new rows will be present in the database following the `ROLLBACK` statement; both rows inserted for `dilbert` and `wally` will be discarded.

row, rows A horizontal unit of storage in a *table*. Each new data record inserted into a table comprises a row; the row in turn is broken into individual *column* values.

scalar, scalar value A single value, such as `'a'`, `123` or `'2008-02-01'`.

scalar subquery, scalar subqueries A scalar subquery is a *subquery* that returns a single column from a single row. Scalar subqueries can be used like columns or anywhere an expression is required, which typically includes the *columns clause* or *WHERE clause* of a `SELECT` statement.

Below, a scalar subquery is used in the columns clause to select the `name` column from the `user_account` table for each row selected from the `email_address` table:

```
SELECT
    email_address.email,
    (SELECT user_account.name FROM user_account WHERE id=1) AS name
FROM email_address WHERE email_address.user_account_id=1
```

```
     email     | name
---------------+----------
 jack@jack.com | jack
```

Selecting an email address by user name, using a scalar subquery in the `WHERE` clause:

```
SELECT email_address.email FROM email_address
WHERE email_address.user_account_id=
    (SELECT id FROM user_account WHERE name='jack')
```

```
    email
---------------
 jack@jack.com
```

Structured Query Language, SQL A special-purpose programming language designed for managing data in relational database management systems (RDBMS).

Originally based upon relational algebra and tuple relational calculus, its scope includes data insert, query, update and delete, schema creation and modification, and data access control.

(via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Sql

subquery A `SELECT` statement embedded in another `SELECT` statement. Data returned from the inner `SELECT` is available for use by the outer.

The subquery is a fundamental capability in SQL that allows so-called *derived tables* to be created; meaning, the rows from a particular `SELECT` statement can be named as a unit of rows within an enclosing `SELECT` that causes it to behave more or less like a plain *table*.

Example:

```
SELECT user_account.name, subq.ad_count FROM
    user_account JOIN
    (SELECT user_account_id, count(id) AS ad_count
    FROM email_address GROUP BY user_account_id) AS subq
    ON user_account.id=subq.user_account_id
```

Subqueries can be placed in a variety of ways inside of an enclosing `SELECT` statement. Three common locations include the *columns clause*, the *WHERE clause*, and the *FROM clause*. The placement of the subquery has an impact on the kind of data the query must return. In standard SQL, subqueries placed within the columns or WHERE clause must be *scalar subqueries*, i.e. queries that return a single value, unless they are evaluated by a boolean aggregation operator such as *IN*, *EXISTS*, `ANY` or `ALL`. A subquery used in the *FROM clause*, on the other hand, can return any number of rows and columns.

Subqueries within the WHERE clause or columns clause are often *correlated subqueries* as well, as they are invoked for each row received in the enclosing query. For a FROM clause subquery, correlation is not an option as the FROM clause is evaluated before the correlatable rows are chosen.

surrogate primary key A *primary key* that is not derived from application data.

(via Wikipedia)

Surrogate primary keys in practice are often integer values generated by database sequences or other incrementing counters, or less commonly global unique identifiers (GUIDs).

See also:

natural primary key

http://en.wikipedia.org/wiki/Surrogate_key

table A fundamental storage component used by relational databases. The table corresponds to what's known as a *relation* in *relational algebra*, and defines a series of *columns*, each of which represents a particular type of data value to be stored in the table. The columns are then organized at the data storage level into a collection of *rows*, each of which corresponds to a unit of data.

table value, rowset An ordered collection of row values, each of the same length and types.

transaction, transactional A transaction comprises a unit of work (not to be confused with SQLAlchemy's *unit of work* pattern, which is similar) performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes:

- To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status.
- To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the programs' outcomes are possibly erroneous.

(via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Database_transaction

ACID Model

commit

rollback

tuple, tuples, row value An ordered collection of typed values, such as `(1, 'ed', 'ed@msn.com')`.

uncorrelated subquery A *subquery* is uncorrelated if the database can execute it in isolation, without referring to the enclosing `SELECT` statement.

```
SELECT user_account.name FROM user_account
WHERE user_account.id IN (SELECT user_account_id FROM email_address)

 name
-------
 jack
 ed
 wendy
```

unique constraint, unique key index A unique key index can uniquely identify each row of data values in a database table. A unique key index comprises a single column or a set of columns in a single database table. No two distinct rows or data records in a database table can have the same data value (or combination of data values) in those unique key index columns if NULL values are not used. Depending on its design, a database table may have many unique key indexes but at most one primary key index.

(via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Unique_key#Defining_unique_keys

WHERE clause A component of the `SELECT` statement which specifies logical criteria to be applied to each row retrieved from the *FROM clause*. The `SELECT` statement discards all rows which do not evaluate to "true" for a given WHERE clause.

Below, we select rows from the `email_address` table, but use the WHERE clause to limit the results to only those rows which refer to email addresses that contain `@gmail.com`:

```
SELECT id, email_address FROM email_address
WHERE email_address LIKE '%@gmail.com'
```

4.2 SQLAlchemy Core / Object Relational Terms

association relationship A two-tiered *relationship* which links two tables together using an association table in the middle. The association relationship differs from a *many to many* relationship in that the many-to-many table is mapped by a full class, rather than invisibly handled by the `sqlalchemy.orm.relationship()` construct as in the case with many-to-many, so that additional attributes are explicitly available.

For example, if we wanted to associate employees with projects, also storing the specific role for that employee with the project, the relational schema might look like:

```
CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE project (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee_project (
    employee_id INTEGER PRIMARY KEY,
    project_id INTEGER PRIMARY KEY,
    role_name VARCHAR(30),
    FOREIGN KEY employee_id REFERENCES employee(id),
    FOREIGN KEY project_id REFERENCES project(id)
)
```

A SQLAlchemy declarative mapping for the above might look like:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key)
    name = Column(String(30))


class Project(Base):
    __tablename__ = 'project'

    id = Column(Integer, primary_key)
    name = Column(String(30))


class EmployeeProject(Base):
    __tablename__ = 'employee_project'

    employee_id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    project_id = Column(Integer, ForeignKey('project.id'), primary_key=True)
    role_name = Column(String(30))

    project = relationship("Project", backref="project_employees")
    employee = relationship("Employee", backref="employee_projects")
```

Employees can be added to a project given a role name:

```
proj = Project(name="Client A")

emp1 = Employee(name="emp1")
```

```
emp2 = Employee(name="emp2")

proj.project_employees.extend([
    EmployeeProject(employee=emp1, role="tech lead"),
    EmployeeProject(employee=emp2, role="account executive")
])
```

See also:

many to many

attribute In Python, a field of an instance or class. Essentially, any time the "." operator is used to access a field from a parent record, you're dealing with attribute access.

Below, the `Car` class has attributes `color` and `model`:

```
class Car(object):
    color = "green"
    model = "Dodge"
```

and attributes are accessed using the "." operator:

```
print("Color: %s" % Car.color)
```

In SQLAlchemy, an ORM *mapped* class is *instrumented* using Python *descriptors* to provide attributes that have additional behaviors used by the mapper, including that changes in value are detected and also that SQL load operations can transparently occur when they are first accessed (known as *lazy loading*).

autocommit This refers to a behavior whereby individual statements are automatically committed to the database after execution, essentially removing the need to explicitly demarcate the beginining and end of a transactional block. Autocommit is something that can take place at many levels and in different ways; some databases will start an interactive SQL session with autocommit implicitly enabled, and others will not, requiring that the user invoke an explicit `COMMIT` statement in order to commit any changes.

When using the Python *DBAPI*, the `connection` object provided by DBAPI is always non-autocommitting by default; that is, the user must call `connection.commit()` in order for the effect of any statements to be committed. Some DBAPIs offer "autocommit" options, but these are not standard.

SQLAlchemy's take on autocommit is that operations which involve executing statements using the Core `Engine` or `Connection` objects are by default autocommitting if the statement represents one that modifies data. If one wants to control the scope of these transactions explicitly, this control is readily available via the `begin()` method. The rationale here is that the Core can be expediently used in a "one-off" style for scripting without the need to deal with transaction demarcation if not needed.

However, when using the ORM `Session` object, the default in modern versions is that the `commit()` method must be called in order to commit the ongoing transaction. The rationale for this is so that the *unit of work* pattern can be used most effectively, where it can safely autoflush data to the database automatically knowing that it's not implicitly permanent, as well as that the explicit commit step provides a clear boundary as to when the ORM-mapped objects should be expired so that they can re-load their state from the database. Ironically, the explicit commit pattern of the `Session` ultimately allows for code that is *more* succinct than if autocommit were turned on, as without it, it's often the case that flushing and expiration must be handled manually.

backref An extension to the *relationship* system whereby two distinct `relationship()` objects can be mutually associated with each other, such that they coordinate in memory as changes occur to either side. The most common way these two relationships are constructed is by using the `relationship()` function explicitly for one side and specifying the `backref` keyword to it so that the other `relationship()` is created automatically. We can illustrate this against the example we've used in *one to many* as follows:

```
class Department(Base):
    __tablename__ = 'department'
```

```
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    employees = relationship("Employee", backref="department")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
```

A backref can be applied to any relationship, including one to many, many to one, and *many to many*.

See also:

relationship

one to many

many to one

many to many

bind, bound This term refers to the association of a connection-producing object, usually an *engine*, with a query-producing object, which in modern usage is usually a *session* object, and in less common usage a *metadata* object.

Most of SQLAlchemy's usage patterns involve dealing with objects that produce SQL queries to be emitted to a database. But it makes a distinction between these objects and objects that represent actual database connections, or a source of database connections.

For example, we can create an ORM `Session` object:

```
>>> from sqlalchemy.orm import Session
>>> session = Session()
```

But if we try to execute a query with it, we'd get an error:

```
>>> session.scalar("select current_timestamp")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/classic/dev/sqlalchemy/lib/sqlalchemy/orm/session.py", line 921, in scalar
    clause, params=params, mapper=mapper, bind=bind, **kw).scalar()
  File "/Users/classic/dev/sqlalchemy/lib/sqlalchemy/orm/session.py", line 912, in execute
    bind = self.get_bind(mapper, clause=clause, **kw)
  File "/Users/classic/dev/sqlalchemy/lib/sqlalchemy/orm/session.py", line 1083, in get_bind
    ', '.join(context)))
sqlalchemy.exc.UnboundExecutionError: Could not
    locate a bind configured on SQL expression or this Session
```

This is because we haven't given this `Session` a source of connectivity. We can make one using `create_engine()` and attaching it using `.bind`:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine("sqlite://")
>>> session.bind = engine
>>> session.scalar("select current_timestamp")
u'2013-02-18 21:13:31'
```

Binding gets more elaborate than this, as a `Session` can be bound to multiple databases at once; some use cases also involve binding the session directly to an individual connection object, rather than to an engine. The practice of using binds with a Core *metadata* object is also something seen commonly, though we've tried to discourage the use of this pattern as it tends to be overused and misunderstood.

cascade The propagation of particular lifecycle events from one mapped instance to another, following along the paths formed by *relationships* between mappings.

An example of the most common cascade is the `save-update` cascade, which states that if an object is associated with a parent, then that object should also be associated with the same *session* as that parent:

```
>>> from sqlalchemy.orm import Session
>>> session = Session()
>>> user_obj = User()
>>> session.add(user_obj)
>>> user_obj in session
True
>>> address_obj = Address()
>>> user_obj.addresses.append(address_obj)
>>> address_obj in session
True
```

Above, we associated an `Address` object with a parent `User` object by appending it to the mapped `User.addresses` collection. As a result, that `Address` object became associated with the same `Session` object as that of the `User`.

The behavior of cascades is customizable, but in most cases the default cascade of `save-update` remains in place.

There are two optional cascades known as `delete` and `delete-orphan` which are also very prominent. These cascades add on the behavior that the child object should also be *deleted* when the parent object is deleted, and additionally that the child object should be deleted when detached from any parent.

The concept of configurable cascade behavior was part of the SQLAlchemy ORM very early on and was inspired by the same configurability in the Hibernate ORM.

See also:

Cascades - in the SQLAlchemy documentation

Configuring delete/delete-orphan Cascade - in the SQLAlchemy documentation

collection In the SQLAlchemy ORM, this refers to a series of objects associated with a parent object, using a *relationship* to manage these associations. A collection corresponds to either a *one to many* or *many to many* relationship, and can be managed in Python by a variety of data types, the most common being the Python `list()`, but also including the Python `set()`, the Python `dict()`, as well as any user-defined type which corresponds to certain interfaces.

When starting out with the SQLAlchemy ORM, we generally stick to plain lists and sets for collections. Dictionaries and custom-build collections are generally for more advanced usage patterns.

See also:

Collection Configuration and Techniques - advanced collection options, in the SQLAlchemy documentation

connection Refers to an active database handle. The term "connection" can refer to different specific constructs; the most fundamental is the "connection" object provided by the Python *DBAPI*. In SQLAlchemy, the DBAPI connection is normally maintained transparently behind a *facade* known as the `Connection` object. This object is obtained from a *engine* object, and has a one-to-one correspondence with a DBAPI connection.

See also:

Engine Configuration - in the SQLAlchemy documentation

Working with Engines and Connections - in the SQLAlchemy documentation

connection pool An object that maintains a series of *connection* objects persistently in memory, allowing individual connections to be *checked out* by a particular application function, used for some period of time, and then *checked in* to the pool when usage of the connection is complete.

The usage of connection pools in SQLAlchemy has two primary purposes:

1. To reduce the latency involved in acquiring a database connection. By maintaining a series of connections in memory, the overhead of the TCP/IP connection as well as the initial negotiation of the client *DBAPI* library with the backend database is incurred only a limited number of times, rather than for all distinct usages of a connection.
2. To place a limit on the number of database connections a single Python process can use at once. SQLAlchemy's default connection pool allows the specification of a *pool size* as well as *max overflow* parameters; the size indicates the largest number of connections that should be held in memory persistently, and the max overflow indicates an optional additional number of connections that may be temporarily procured on top of the base size.

The SQLAlchemy *engine* object maintains a reference to a connection pool where it retrieves and stores DBAPI connections - in most cases this pool is an instance of `sqlalchemy.pool.QueuePool`. Connection pooling can be disabled for a particular engine by replacing the pool implementation with the so-called `sqlalchemy.pool.NullPool`, which has the same interface as a pool but doesn't actually maintain connections persistently.

Note that SQLAlchemy's built-in pooling is only one style of pooling, known as *application level pooling*. An architecture can also use *pool middleware*, i.e., a server that runs separately and mediates connectivity between one or more applications and a database backend. The PgBouncer product is one such middleware service designed for usage with Postgresql.

See also:

Connection Pooling

DBAPI DBAPI is shorthand for the phrase "Python Database API Specification". This is a widely used specification within Python to define common usage patterns for all database connection packages. The DBAPI is a "low level" API which is typically the lowest level system used in a Python application to talk to a database. SQLAlchemy's *dialect* system is constructed around the operation of the DBAPI, providing individual dialect classes which service a specific DBAPI on top of a specific database engine; for example, the `create_engine()` URL `postgresql+psycopg2://@localhost/test` refers to the `psycopg2` DBAPI/dialect combination, whereas the URL `mysql+mysqldb://@localhost/test` refers to the `MySQL for Python` DBAPI/dialect combination.

See also:

PEP 249 - Python Database API Specification v2.0: http://www.python.org/dev/peps/pep-0249/

declarative An API included with the SQLAlchemy ORM that in modern usage serves as the primary system used to configure the ORM. The central idea of the declarative system is that one defines a class to be *mapped*, and then applies to this class a series of directives which denote the *table metadata* to be associated with this class, which establishes the table(s) and columns that this class will be associated with when the ORM performs queries.

The declarative system provides a relatively concise and very extensible series of patterns allowing not just for basic class mapping, but also allowing the construction of repeatable and composable mapping patterns using custom base classes, abstract classes, and mixins.

See also:

Object Relational Tutorial

Declarative

descriptor, descriptors In Python, a descriptor is an object attribute with "binding behavior" whose attribute access has been overridden by methods in the descriptor protocol. Those methods are __get__(), __set__(), and __delete__(). If any of those methods are defined for an object, it is said to be a descriptor.

In SQLAlchemy, descriptors are used heavily in order to provide attribute behavior on mapped classes. When a class is mapped as such:

```
class MyClass(Base):
    __tablename__ = 'foo'

    id = Column(Integer, primary_key=True)
    data = Column(String)
```

The `MyClass` class will be *mapped* when its definition is complete, at which point the `id` and `data` attributes, starting out as `sqlalchemy.schema.Column` objects, will be replaced by the *instrumentation* system with customized descriptor objects, providing special behavior for the `__get__()`, `__set__()` and `__delete__()` methods. The descriptors (for the curious, they are instances of `sqlalchemy.orm.attributes.InstrumentedAttribute`, though this detail is generally transparent) will generate a SQL expression when used at the class level:

```
>>> print MyClass.data == 5
data = :data_1
```

When used at the instance level, these descriptors help to keep track of changes to values, and also *lazy load* unloaded values and collections from the database when the attribute is accessed.

detached This describes one of the four major object states which an object can have within a *session*; a detached object is an object that has a database identity (i.e. a primary key) but is not associated with any session. An object that was previously *persistent* and was removed from its session either because it was expunged, or the owning session was closed, moves into the detached state. The detached state is generally used when objects are being moved between sessions or when being moved to/from an external object cache.

See also:

Quickie Intro to Object States - in the SQLAlchemy documentation

engine An object that provides a source of database connectivity. The `Engine` object maintains a *connection pool*, which keeps track of a series of *DBAPI* connection objects, as well as a *dialect*, which keeps track of all the information known about the particular kind of database and Python driver being used by this particular engine. An `Engine` is created using the `create_engine()` factory function, and a database connection can be requested from the `Engine` using the `connect()` method:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine("postgresql://scott:tiger@localhost/test")
>>> connection = engine.connect()
>>> connection.scalar("SELECT now()")
datetime.datetime(2013, 2, 18, 18, 26, 37)
>>> connection.close()
```

While the above pattern illustrates a literal, rudimentary use of `Engine`, it's normally used in a more abstracted way than the above. When dealing with the SQLAlchemy ORM, the `Engine` is usually *bound* to an ORM *session* object when the program starts, where it then remains hidden as a source of connectivity for that session.

The primary facade for a database. An `Engine` manages a pool of database connections and provides methods to execute SQL statements and fetch result sets.

See also:

Engine Configuration - in the SQLAlchemy documentation

Working with Engines and Connections - in the SQLAlchemy documentation

flush The operation by which a *session* emits INSERT, UPDATE and DELETE statements to the database in response to the accumulation of a series of in-memory changes to objects. The flush operation is a key component of the *unit of work* pattern, and is normally invoked before the `Session` emits a new SELECT statement, as well as right before the `Session` commits a transaction.

See also:

Flushing - in the SQLAlchemy documentation

identity map A mapping between Python objects and their database identities. The identity map is a collection that's associated with an ORM *session* object, and maintains a single instance of every database object keyed to its identity. The advantage to this pattern is that all operations which occur for a particular database identity are transparently coordinated onto a single object instance. When using an identity map in conjunction with an *isolated* transaction, having a reference to an object that's known to have a particular primary key can be considered from a practical standpoint to be a proxy to the actual database row.

See also:

Martin Fowler - Identity Map - http://martinfowler.com/eaaCatalog/identityMap.html

instance Refers to an instantiated object, that is, the result of calling the constructor of a Python class.

We use this term to specify that we are dealing with a stateful Python object, rather than the class. Suppose we have a class called `User`:

```
class User(object):
    def __init__(self, name):
        self.name = name
```

The above Python code represents only the *class* `User`, and not an actual instance. The instance refers to when we construct a `User`, and in this case assign to it a `.name` *attribute*:

```
my_user = User('some user')
```

The SQLAlchemy ORM deals heavily with user-defined classes and instances of those classes; therefore throughout its documentation as well as its source code, it's important that we keep straight whether we're dealing with a class or an instance of one.

instrumentation, instrumented Instrumentation refers to the process of augmenting the functionality and attribute set of a particular class. Ideally, the behavior of an instrumented class should remain close to a regular class, except that additional behviors and features are made available. The SQLAlchemy *mapping* process, among other things, adds database-enabled *descriptors* to a mapped class which each represent a particular database column or relationship to a related class.

lazy load, lazy loads, lazy loading In object relational mapping, a "lazy load" refers to an attribute that does not contain its database-side value for some period of time, typically when the object is first loaded. Instead, the attribute receives a *memoization* that causes it to go out to the database and load its data when it's first used. Using this pattern, the complexity and time spent within object fetches can sometimes be reduced, in that attributes for related tables don't need to be addressed immediately.

See also:

Martin Fowler - Lazy Load - http://martinfowler.com/eaaCatalog/lazyLoad.html

N plus one problem

many to many A style of `sqlalchemy.orm.relationship()` which links two tables together via an intermediary table in the middle. Using this configuration, any number of rows on the left side may refer to any number of rows on the right, and vice versa.

A schema where employees can be associated with projects:

```
CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE project (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee_project (
    employee_id INTEGER PRIMARY KEY,
    project_id INTEGER PRIMARY KEY,
    FOREIGN KEY employee_id REFERENCES employee(id),
    FOREIGN KEY project_id REFERENCES project(id)
)
```

Above, the `employee_project` table is the many-to-many table, which naturally forms a composite primary key consisting of the primary key from each related table.

In SQLAlchemy, the `sqlalchemy.orm.relationship()` function can represent this style of relationship in a mostly transparent fashion, where the many-to-many table is specified using plain table metadata:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key)
    name = Column(String(30))

    projects = relationship(
        "Project",
        secondary=Table('employee_project', Base.metadata,
                    Column("employee_id", Integer, ForeignKey('employee.id'),
                                primary_key=True),
                    Column("project_id", Integer, ForeignKey('project.id'),
                                primary_key=True)
                ),
        backref="employees"
        )

class Project(Base):
    __tablename__ = 'project'

    id = Column(Integer, primary_key)
    name = Column(String(30))
```

Above, the `Employee.projects` and back-referencing `Project.employees` collections are defined:

```
proj = Project(name="Client A")

emp1 = Employee(name="emp1")
emp2 = Employee(name="emp2")

proj.employees.extend([emp1, emp2])
```

See also:

association relationship

relationship

one to many

many to one

many to one A style of `relationship()` which links a foreign key in the parent mapper's table to the primary key of a related table. Each parent object can then refer to exactly zero or one related object.

The related objects in turn will have an implicit or explicit *one to many* relationship to any number of parent objects that refer to them.

An example many to one schema (which, note, is identical to the *one to many* schema):

```
CREATE TABLE department (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    dep_id INTEGER REFERENCES department(id)
)
```

The relationship from `employee` to `department` is many to one, since many employee records can be associated with a single department. A SQLAlchemy mapping might look like:

```
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
    department = relationship("Department")
```

See also:

relationship

one to many

backref

mapped, mapper, mapping We say a class is "mapped" when it has been passed through the `sqlalchemy.orm.mapper()` function. This process associates the class with a database table or other *selectable* construct, so that instances of it can be persisted and loaded using a *session* object.

Modern usage of the SQLAlchemy ORM typically "maps" classes using the *declarative* system, which provides a relatively concise and very extensible series of patterns allowing classes to be mapped. The declarative system actually rides on top of the so-called *Classical Mappings* system, which is more fundamental and less automated. Early versions of SQLAlchemy only featured the classical mapping system.

metadata, table metadata A collection of related `Table` objects. These objects collected together may define `ForeignKey` objects which refer to other tables as dependencies. The full collection of tables can be created and dropped in a target database schema en masse.

See also:

Describing Databases with MetaData - in the SQLAlchemy documentation

N plus one problem The N plus one problem is a common side effect of the *lazy load* pattern, whereby an application wishes to iterate through a related attribute or collection on each member of a result set of objects, where that attribute or collection is set to be loaded via the lazy load pattern. The net result is that a SELECT statement is emitted to load the initial result set of parent objects; then, as the application iterates through each member, an additional SELECT statement is emitted for each member in order to load the related attribute or collection for that member. The end result is that for a result set of N parent objects, there will be N + 1 SELECT statements emitted.

The N plus one problem is alleviated using *eager loading*.

one to many A style of `relationship()` which links the primary key of the parent mapper's table to the foreign key of a related table. Each unique parent object can then refer to zero or more unique related objects.

The related objects in turn will have an implicit or explicit *many to one* relationship to their parent object.

An example one to many schema (which, note, is identical to the *many to one* schema):

```
CREATE TABLE department (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    dep_id INTEGER REFERENCES department(id)
)
```

The relationship from `department` to `employee` is one to many, since many employee records can be associated with a single department. A SQLAlchemy mapping might look like:

```
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    employees = relationship("Employee")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
```

See also:

relationship

many to one

backref

pending This describes one of the four major object states which an object can have within a *session*; a pending object is a new object that doesn't have any database identity, but has been recently associated with a session. When the session emits a flush and the row is inserted, the object moves to the *persistent* state.

See also:

Quickie Intro to Object States - in the SQLAlchemy documentation

persistent This describes one of the four major object states which an object can have within a *session*; a persistent object is an object that has a database identity (i.e. a primary key) and is currently associated with a session. Any object that was previously *pending* and has now been inserted is in the persistent state, as is any object

that's been loaded by the session from the database. When a persistent object is removed from a session, it is known as *detached*.

See also:

Quickie Intro to Object States - in the SQLAlchemy documentation

reflection The process of constructing SQLAlchemy `Table` objects in an automated or semi-automated fashion, where information about tables, columns and constraints are loaded from an existing database's internal catalogs in order to compose in-memory structures representing a schema.

See also:

metadata_reflection

relationship, relationships A connecting unit between two mapped classes, corresponding to some relationship between the two tables in the database.

The relationship is defined using the SQLAlchemy function `relationship()`. Once created, SQLAlchemy inspects the arguments and underlying mappings involved in order to classify the relationship as one of three types: *one to many*, *many to one*, or *many to many*. With this classification, the relationship construct handles the task of persisting the appropriate linkages in the database in response to in-memory object associations, as well as the job of loading object references and collections into memory based on the current linkages in the database.

See also:

Relationship Configuration - in the SQLAlchemy documentation

scoped session A helper object intended to provide a *registry* of *session* objects, allowing an application to refer to the registry as a global variable which provides access to a contextually appropriate session object.

The scoped session object is an optional construct often used with web applications.

See also:

Session

Contextual/Thread-local Sessions - an in-depth introduction to the `sqlalchemy.orm.scoped_session` object, in the SQLAlchemy documentation

selectable Refers to the SQLAlchemy analogue for a "relation" in relational algebra, which is any object that represents a series of rows in a database. "Selectable" actually refers in the API to objects that extend from the `sqlalchemy.sql.expression.Selectable` class, and refers to such row-representing constructs as the `Table`, the `Join`, and the `Select` construct.

Session The container or scope for ORM database operations. Sessions load instances from the database, track changes to mapped instances and persist changes in a single unit of work when flushed.

See also:

Using the Session

sessionmaker A *factory* for *session* objects. The `sessionmaker` construct basically allows a series of parameters to be associated with a `Session` constructor.

In reality, the sessionmaker is just slightly more elaborate than a simple function. An expression like this:

```
from sqlalchemy.orm import sessionmaker
my_session = sessionmaker(bind=engine, autoflush=False)
```

is conceptually very similar to the following:

```
from sqlalchemy.orm import Session
my_session = lambda: Session(bind=engine, autoflush=False)
```

See also:

Session

scoped session

threadlocal A shared data structure whose data members are visible only to the thread which set them. The concept of "thread local" in Python is normally provided by the `threading.local` construct.

See also:

http://docs.python.org/2/library/threading.html#threading.local

transient This describes one of the four major object states which an object can have within a *session*; a transient object is a new object that doesn't have any database identity and has not been associated with a session yet. When the object is added to the session, it moves to the *pending* state.

See also:

Quickie Intro to Object States - in the SQLAlchemy documentation

unit of work This pattern is where the system transparently keeps track of changes to objects and periodically flushes all those pending changes out to the database. SQLAlchemy's Session implements this pattern fully in a manner similar to that of Hibernate.

See also:

Martin Fowler - Unit of Work - http://martinfowler.com/eaaCatalog/unitOfWork.html

Using the Session - in the SQLAlchemy documentation

CHAPTER

FIVE

FURTHER READING

Credit to Jason Kirtland for assembling this list.

- Celko, J. (2010) *SQL for Smarties.* Morgan Kaufmann.

  A sprawling book with many gems of SQL knowledge. This book is now up to its fourth edition and seems to have been modified quite a bit.

- The SQL92 Standard

  Available as a .txt file from http://en.wikipedia.org/wiki/SQL-92

- Harrington, J. (2003) *SQL Clearly Explained.* Morgan Kaufmann.

  Exactly what the title claims.

- Schmidt, B. (1999) *Data Modeling for Information Professionals.* Prentice Hall PTR.

  A fantastic resource for anyone in any profession who needs to think critically about information and its structure.

- Nock, C. (2004) *Data Access Patterns.* Addison-Wesley.

- Fowler, M. (2002) *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

- Schmidt, D., et al. (2000) *Pattern-Oriented Software Architecture volume 2, Patterns for Concurrent and Networked Objects.* Wiley.

  These three delve into the the mechanics of data access and strategies for concurrency. *Patterns of Enterprise Application Architecture* was a primary inspiration for the creation of SQLAlchemy itself. The blog post *Patterns Implemented by SQLAlchemy* (http://techspot.zzzeek.org/2012/02/07/patterns-implemented-by-sqlalchemy) details this.

- Hay, D. (1996) *Data Model Patterns, Conventions of Thought.* Dorset House.

- Fowler, M. (1997) *Analysis Patterns.* Addison-Wesley.

  Both books provide good schema advice and domain knowledge for classic topics such as accounting.

N

O

P

Q

R

S

T

U

W