SQLAlchemy Glossary

attribute
In Python, a field of an instance or class. In SQLAlchemy, managed instance attributes are instrumented to detect changes, and class attributes can be used in the construction of queries.

autocommit
A style of SQL execution in which each statement is transparently wrapped in its own transaction. Autocommit is often used for interactive SQL command line prompts and it is the default behavior of SQLAlchemy if no explicit transaction is in effect.

bind
The association between a database and a SQLAlchemy component such as a table or ORM session. Components which are bound are linked to a single database and can act upon it implicitly. Unbound components can be used with any number of databases but must explicitly combined: "act upon database ‘A’, now act upon database ‘B’."

cascade
The propagation of events from one mapped instance to another. The cascade follows the path defined by relations between the mappings. Cascaded events communicate session state. For example, adding a lead instance to a session will add all associated instances as well.

collection
In Python, a container class such as a list, set, or dict. In SQLAlchemy relations, the same, except the collection is automatically filled with instances retrieved from the database.

connection
An active link to a database. In SQLAlchemy, connections are managed in a pool and idle connections are reused for new tasks. An active connection has a transactional state.

column
1. a database column, as defined within a CREATE TABLE statement

2. a SQLAlchemy Column construct, which is a data structure that stores information about the name of a database column, its constraints, data type, and information on its default value.

detached
An instance which is not present in any session, but whose state information is present in the database. A detached instance may have further pending changes on it which will only be persisted if the instance is updated into a session and then flushed.

engine
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.

explicit execution
Applying a SQLAlchemy action or statement to a specific database or connection.

flush
To wash clean. In the SQLAlchemy ORM, a session flush will send changes in the session's Unit of Work to the database and begin a new unit of work.

identity map
A per-session, one-to-one mapping between Python instances and database identity. Ensures that only one mapped instance exists at a time, no matter how how it is queried or associated.

implicit execution
Application of a bound SQLAlchemy action or statement, affects the database specified in the bind. Provides a lightweight syntax when only a single database is required.

instance
The result of calling a Python class constructor; a single, unique Python object.

instrumentation
The injection of an observer into a method or attribute. SQLAlchemy uses instrumentation to detect changes made to managed attributes and track changes in collection membership. Changes raise events which can cascade to related instances.

mapper
An object which translates database rows to and from instances of a class. Mappers define which columns will be translated to object attributes, and how foreign key relationships will be translated to collection-holding attributes. A mapper installs instrumentation on the Python class to manage mapped attributes.

MetaData
A collection of related Table objects. Tables collected together may define their ForeignKeys as simple strings, and can be created and dropped in the database *en masse*.

orphan
A mapped instance with a severed link to a collection or parent object.

pending
An instance which has been saved into a session but not yet persisted to the database.

persistent
An instance which is present in a session and in the database.

query
1. A SQL statement which is processed by a database to return results.

2. A SQLAlchemy ORM object which defines search criterion and returns mapped instances.

threadlocal
A shared data structure whose data members are visible only to the thread which set them.

reflection
The process of constructing SQLAlchemy Table objects programatically at runtime by querying a live database's system tables for column and key definitions.

relation
1. 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.

2. In SQLAlchemy, the junction of two mapped classes, or of a mapped class to itself. The relation usually corresponds to a foreign key relationship between two tables or selectables.

scoped_session
A front end for sessionmaker which provides a “global” registry of sessions, each mapped to the current thread.

selectable
What relational algebra refers to as a “relation”, SQLAlchemy refers to as a selectable. A table, subquery, or any other table-valued SQL expression.

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.

session transaction
ORM-level transaction. Session activity may span multiple databases, and the session transaction coordinates a connection-level transaction for each. Database features such as save points and two-phase transactions are also supported.

sessionmaker
An optional, configurable factory object used to create new Session instances using a chosen set of construction arguments.

table
1. A database table, defined by a CREATE TABLE statement.

2. A SQLAlchemy Table construct, which is a data structure that stores information about the name of a database table, its columns and other constraints.

transient
An instance of a mapped class which has not been saved into a session or loaded from the database.

transaction
A unit of work within the database specific to an specific database connection. All statements take effect together or not at all. A committed transaction changes the database permanently, and a rolled back transaction makes no changes. In SQLAlchemy, transactions are available on the connection, engine and session levels.

Unit of Work (or UOW)
The bundling together of all pending mapped instance creations, modifications and deletions. The workhorse behind an ORM session flush, the Unit of Work translates un-flushed session activity into a properly ordered series of INSERT, UPDATE and DELETE statements.

Relational Cheat Sheet

DML

Data Manipulation Language; the SQL commands that manipulate data. For example, SELECT, INSERT, UPDATE and DELETE.

DDL

Data Definition Language; the SQL commands that define a schema. For example, CREATE TABLE, DROP TABLE, ALTER TABLE.

join (or 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 * FROM users JOIN addresses ON users.id = addresses.user_id

 id | name  | id |     email     | user_id
----+-------+----+---------------+---------
  1 | jack  |  1 | jack@jack.com |       1
  2 | ed    |  2 | ed@yahoo.com  |       2
  2 | ed    |  3 | ed@msn.com    |       2
  3 | wendy |  4 | wendy@nyt.com |       3
```

left outer join

Combines the rows of two tables. Using an ON criteria, compares each row in the first table listed—the “left” table—against each row in the right table. Any matches are returned like an inner join. If a left row matches no right rows, returns a row containing the columns of the left row plus NULLs for every column in the right table.

```
SELECT * FROM users
LEFT OUTER JOIN addresses ON users.id = addresses.user_id

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

right outer join

Like a left outer join, except the tables 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. In SQLAlchemy, outer joins are left outer joins.

scalar value

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

row value

An ordered collection of typed values, such as (1, 'ed', 'ed@msn.com'). Also referred to as a tuple.

table value

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

subquery (or subselect)

A SELECT statement embedded in another SELECT statement. Data returned from the inner SELECT is available for use by the outer. Subqueries can be used almost anywhere in a query, but are typically used as columns, in the FROM and WHERE clauses.

scalar subquery

A scalar subquery is a SELECT that returns a single column from a single row. Scalar subqueries can be used like columns or anywhere an expression is required:

```
SELECT users.name FROM users WHERE id=1

 name
------
 jack

SELECT addresses.email, (SELECT users.name FROM users WHERE id=1)
FROM addresses WHERE addresses.user_id=1

     email     | ?column?
---------------+----------
 jack@jack.com | jack
```

They are also useful in the WHERE clause of a query:

```
SELECT addresses.email FROM addresses
WHERE addresses.user_id=(SELECT id FROM users WHERE name='jack')

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

uncorrelated subquery

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

```
SELECT users.name FROM users
WHERE users.id IN (SELECT user_id FROM addresses)

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

correlated subquery

A subquery is correlated if it depends on data in the enclosing SELECT.

```
SELECT users.name, addresses.email
FROM users
JOIN addresses ON users.id=addresses.user_id
WHERE addresses.id = (SELECT MIN(a.id) FROM addresses AS a
                      WHERE a.user_id=users.id)

 name  |     email
-------+---------------
 jack  | jack@jack.com
 ed    | ed@yahoo.com
 wendy | wendy@nyt.com
```

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 addresses
WHERE user_id IN (1, 2)

     email
---------------
 jack@jack.com
 ed@yahoo.com
 ed@msn.com
```

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

```
SELECT email FROM addresses
WHERE user_id IN (SELECT id FROM users WHERE name='jack' OR name='ed')

   email
---------------
 jack@jack.com
 ed@yahoo.com
 ed@msn.com
```

EXISTS operator

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

```
SELECT name FROM users
WHERE EXISTS (SELECT * FROM addresses WHERE addresses.user_id=users.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 complete expression and can be combined normally with other criteria in a WHERE clause.

```
SELECT name FROM users
WHERE EXISTS (SELECT * FROM addresses WHERE addresses.user_id=users.id)
  AND name='ed'

 name
------
 ed
```

single table inheritance

Columns for all classes in an inheritance hierarchy are stored in a single table. A discriminator column indicates which class a given row represents. Columns not needed by a particular class are left empty.

```
 id | type  | amount |    date    | cnum | expiry_year | expiry_mon
----+-------+--------+------------+------+-------------+------------
  1 | check | 100.00 | 2008-02-01 |   12 |             |
  2 | ccard |  50.75 | 2008-02-02 |      |        2010 |          2
```

joined table inheritance

Columns for classes in an inheritance hierarchy are stored in one table per class. The tables are joined together to represent an instance– columns for the instance's types are combined with columns of its super class and so on.

The primary key of the base class in the hierarchy is shared among all of class tables. The base class table also contains a discriminator column to identify the type of any given row.

```
CREATE TABLE payment (
  id SERIAL PRIMARY KEY,
  type VARCHAR(16) NOT NULL,
  amount NUMERIC(10,2),
  "date" DATE )

CREATE TABLE check_payment (
  id INTEGER PRIMARY KEY REFERENCES payment (id),
  cnum INTEGER )

CREATE TABLE ccard_payment (
  id INTEGER PRIMARY KEY REFERENCES payment (id),
  expiry_year INTEGER,
  expiry_mon INTEGER )

INSERT INTO payment (type, amount, "date")
  VALUES ('check', 100.0, '2008-02-01')

INSERT INTO check_payment VALUES (1, 12)

INSERT INTO payment (type, amount, "date")
  VALUES ('ccard', 50.75, '2008-02-02')

INSERT INTO ccard_payment VALUES (2, 2010, 2)

SELECT * FROM payment
  NATURAL LEFT JOIN check_payment
  NATURAL LEFT JOIN ccard_payment

 id | type  | amount |    date    | cnum | expiry_year | expiry_mon
----+-------+--------+------------+------+-------------+------------
  1 | check | 100.00 | 2008-02-01 |   12 |             |
  2 | ccard |  50.75 | 2008-02-02 |      |        2010 |          2

SELECT * FROM payment NATURAL JOIN check_payment WHERE type='check'

 id | type  | amount |    date    | cnum
----+-------+--------+------------+------
  1 | check | 100.00 | 2008-02-01 |   12

SELECT * FROM payment NATURAL JOIN ccard_payment WHERE type='ccard'

 id | type  | amount |    date    | expiry_year | expiry_mon
----+-------+--------+------------+-------------+------------
  2 | ccard |  50.75 | 2008-02-02 |        2010 |          2
```

concrete table inheritance

Columns for classes in an inheritance hierarchy are stored in one table per class. Each table contains the full set of columns used by its class, and primary key values are not unique among tables. The tables are fully independent.

```
CREATE TABLE check_payment (
  id SERIAL PRIMARY KEY,
  amount NUMERIC(10,2),
  "date" DATE,
  cnum INTEGER )

 CREATE TABLE ccard_payment (
  id SERIAL PRIMARY KEY,
  amount NUMERIC(10,2),
  "date" DATE,
  expiry_year INTEGER,
  expiry_mon INTEGER )

INSERT INTO check_payment (amount, "date", cnum)
  VALUES (100.0, '2008-02-01', 12)

INSERT INTO ccard_payment (amount, "date", expiry_year, expiry_mon)
  VALUES (50.75, '2008-02-02', 2010, 2)

SELECT * FROM check_payment

 id | amount |    date    | cnum
----+--------+------------+------
  1 | 100.00 | 2008-02-01 |   12

SELECT * FROM ccard_payment

 id | amount |    date    | expiry_year | expiry_mon
----+--------+------------+-------------+------------
  1 |  50.75 | 2008-02-02 |        2010 |          2
```

association table (or join table, or many2many table)

An intermediary table modeling a many-to-many relationship. Given

```
CREATE TABLE cars (
   car_id INTEGER PRIMARY KEY,
   model VARCHAR(100) )

CREATE TABLE colors (
   color_id INTEGER PRIMARY KEY,
   name VARCHAR(100) )
```

and the relationship

```
"car models are available in multiple colors"
```

the relation can be modeled with a two-column table.

```
CREATE TABLE car_colors (
  car_id INTEGER REFERENCES cars (car_id),
  color_id INTEGER REFERENCES colors (color_id),
  PRIMARY KEY (car_id, color_id) )
```

The naming convention 'cars2colors' is common for these types of tables.

properties of a relation

A relationship between two tables that is further qualified by information specific to each pair of linked rows. Given

```
CREATE TABLE cars (
   car_id INTEGER PRIMARY KEY,
   model VARCHAR(100) )

CREATE TABLE colors (
   color_id INTEGER PRIMARY KEY,
   name VARCHAR(100) )
```

and the relationship

```
"car models are available in multiple colors, each for a limited time-span"
```

the relation can be modeled with an association table containing additional columns.

```
CREATE TABLE car_colors (
  car_id INTEGER REFERENCES cars (car_id),
  color_id INTEGER REFERENCES colors (color_id),
  available_starting DATE NOT NULL,
  available_ending DATE NOT NULL,
  PRIMARY KEY (car_id, color_id) )
```

Reading List

- 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.

- 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.

- Nock, C. (2004) *Data Access Pattern*s. 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.

- Celko, J. (2004)*Trees and Hierarchies in SQL for Smarties.* Morgan Kaufmann.
  Covers many strategies for efficiently modeling trees and hierarchies

- Celko, J. (2005) *SQL for Smarties.* Morgan Kaufmann.
  A sprawling book with many gems of SQL knowledge.

- *The SQL92 Standard*
  Available as a .txt file from http://en.wikipedia.org/wiki/SQL-92

Web Site

http://www.sqlalchemy.org/

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
```

IRC Channel

#sqlalchemy on the Freenode network

Presenters

Michael Bayer 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.

Jason Kirtland is an independent software developer and an enthusiastic user of Python. Jason lives in Portland, Oregon and, on rainy days, hacks on SQLAlchemy.

Jonathan Ellis recently left Mozy, Inc., where he wrote a storage system similar to Amazon S3 to store petabytes of backup data. He is the author of the SqlSoup extension to SQLAlchemy. Jonathan lives in Utah with his wife and two children.