SQLAlchemy

Michael Bayer

SQLAlchemy is a database toolkit and object-relational mapping (ORM) system for the Python programming language, first introduced in 2005. From the beginning, it has sought to provide an end-to-end system for working with relational databases in Python, using the Python Database API (DBAPI) for database interactivity. Even in its earliest releases, SQLAlchemy's capabilities attracted a lot of attention. Key features include a great deal of fluency in dealing with complex SQL queries and object mappings, as well as an implementation of the "unit of work" pattern, which provides for a highly automated system of persisting data to a database.

Starting from a small, roughly implemented concept, SQLAlchemy quickly progressed through a series of transformations and reworkings, turning over new iterations of its internal architectures as well as its public API as the userbase continued to grow. By the time version 0.5 was introduced in January of 2009, SQLAlchemy had begun to assume a stable form that was already proving itself in a wide variety of production deployments. Throughout 0.6 (April, 2010) and 0.7 (May, 2011), architectural and API enhancements continued the process of producing the most efficient and stable library possible. As of this writing, SQLAlchemy is used by a large number of organizations in a variety of fields, and is considered by many to be the de facto standard for working with relational databases in Python.

20.1. The Challenge of Database Abstraction

The term "database abstraction" is often assumed to mean a system of database communication which conceals the majority of details of how data is stored and queried. The term is sometimes taken to the extreme, in that such a system should not only conceal the specifics of the relational database in use, but also the details of the relational structures themselves and even whether or not the underlying storage is relational.

The most common critiques of ORMs center on the assumption that this is the primary purpose of such a tool—to "hide" the usage of a relational database, taking over the task of constructing an interaction with the database and reducing it to an implementation detail. Central to this approach of concealment is that the ability to design and query relational structures is taken away from the developer and instead handled by an opaque library.

Those who work heavily with relational databases know that this approach is entirely impractical. Relational structures and SQL queries are vastly functional, and comprise the core of an application's design. How these structures should be designed, organized, and manipulated in queries varies not just on what data is desired, but also on the structure of information. If this utility is concealed, there's little point in using a relational database in the first place.

The issue of reconciling applications that seek concealment of an underlying relational database with the fact that relational databases require great specificity is often referred to as the "object-relational impedance mismatch" problem. SQLAlchemy takes a somewhat novel approach to this problem.

SQLAlchemy's Approach to Database Abstraction

SQLAlchemy takes the position that the developer must be willing to consider the relational form of his or her data. A system which pre-determines and conceals schema and query design decisions marginalizes the usefulness of using a relational database, leading to all of the classic problems of impedance mismatch.

At the same time, the implementation of these decisions can and should be executed through high-level patterns as much as possible. Relating an object model to a schema and persisting it via SQL queries is a highly repetitive task. Allowing tools to automate these tasks allows the development of an application that's more succinct, capable, and efficient, and can be created in a fraction of the time it would take to develop these operations manually.

To this end, SQLAlchemy refers to itself as a *toolkit*, to emphasize the role of the developer as the designer/builder of all relational structures and linkages between those structures and the application, not as a passive consumer of decisions made by a library. By exposing relational concepts, SQLAlchemy embraces the idea of "leaky abstraction", encouraging the developer to tailor a custom, yet fully automated, interaction layer between the application and the relational database. SQLAlchemy's innovation is the extent to which it allows a high degree of automation with little to no sacrifice in control over the relational database.

20.2. The Core/ORM Dichotomy

Central to SQLAlchemy's goal of providing a toolkit approach is that it exposes every layer of database interaction as a rich API, dividing the task into two main categories known as *Core* and *ORM*. The Core includes Python Database API (DBAPI) interaction, rendering of textual SQL statements understood by the database, and schema management. These features are all presented as public APIs. The ORM, or object-relational mapper, is then a specific library built on top of the Core. The ORM provided with SQLAlchemy is only one of any number of possible object abstraction layers that could be built upon the Core, and many developers and organizations build their applications on top of the Core directly.

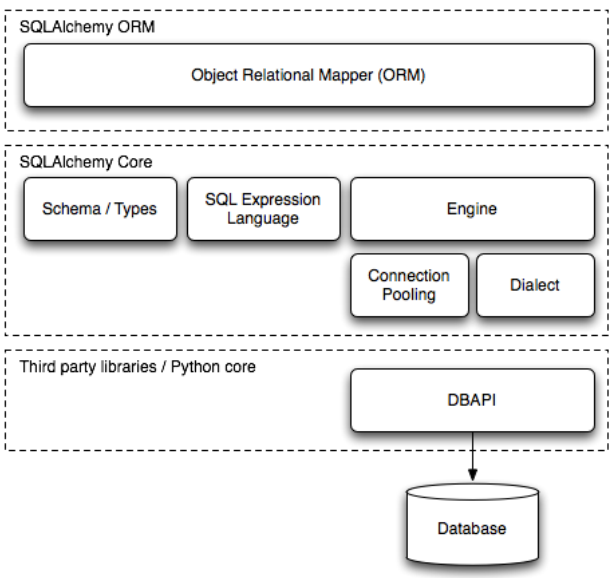

Figure 20.1: SQLAlchemy layer diagram

The Core/ORM separation has always been SQLAlchemy's most defining feature, and it has both pros and cons. The explicit Core present in SQLAlchemy leads the ORM to relate database-mapped class attributes to a structure known as a `Table`, rather than directly to their string column names as expressed in the database; to produce a SELECT query

using a structure called `select`, rather than piecing together object attributes directly into a string statement; and to receive result rows through a facade called `ResultProxy`, which transparently maps the `select` to each result row, rather than transferring data directly from a database cursor to a user-defined object.

Core elements may not be visible in a very simple ORM-centric application. However, as the Core is carefully integrated into the ORM to allow fluid transition between ORM and Core constructs, a more complex ORM-centric application can "move down" a level or two in order to deal with the database in a more specific and finely tuned manner, as the situation requires. As SQLAlchemy has matured, the Core API has become less explicit in regular use as the ORM continues to provide more sophisticated and comprehensive patterns. However, the availability of the Core was also a contributor to SQLAlchemy's early success, as it allowed early users to accomplish much more than would have been possible when the ORM was still being developed.

The downside to the ORM/Core approach is that instructions must travel through more steps. Python's traditional C implementation has a significant overhead penalty for individual function calls, which are the primary cause of slowness in the runtime. Traditional methods of ameliorating this include shortening call chains through rearrangement and inlining, and replacing performance-critical areas with C code. SQLAlchemy has spent many years using both of these methods to improve performance. However, the growing acceptance of the PyPy interpreter for Python may promise to squash the remaining performance problems without the need to replace the majority of SQLAlchemy's internals with C code, as PyPy vastly reduces the impact of long call chains through just-in-time inlining and compilation.

20.3. Taming the DBAPI

At the base of SQLAlchemy is a system for interacting with the database via the DBAPI. The DBAPI itself is not an actual library, only a specification. Therefore, implementations of the DBAPI are available for a particular target database, such as MySQL or PostgreSQL, or alternatively for particular non-DBAPI database adapters, such as ODBC and JDBC.

The DBAPI presents two challenges. The first is to provide an easy-to-use yet full-featured facade around the DBAPI's rudimentary usage patterns. The second is to handle the extremely variable nature of specific DBAPI implementations as well as the underlying database engines.

The Dialect System

The interface described by the DBAPI is extremely simple. Its core components are the DBAPI module itself, the connection object, and the cursor object—a "cursor" in database parlance represents the context of a particular statement and its associated results. A simple interaction with these objects to connect and retrieve data from a database is as follows:

```
connection = dbapi.connect(user="user", pw="pw", host="host")
cursor = connection.cursor()
cursor.execute("select * from user_table where name=?", ("jack",))
print "Columns in result:", [desc[0] for desc in cursor.description]
for row in cursor.fetchall():
    print "Row:", row
cursor.close()
connection.close()
```

SQLAlchemy creates a facade around the classical DBAPI conversation. The point of entry to this facade is the `create_engine` call, from which connection and configuration information is assembled. An instance of `Engine` is produced as the result. This object then represents the gateway to the DBAPI, which itself is never exposed directly.

For simple statement executions, `Engine` offers what's known as an *implicit execution* interface. The work of acquiring and closing both a DBAPI connection and cursor are handled behind the scenes:

```
engine = create_engine("postgresql://user:pw&#64;host/dbname")
result = engine.execute("select * from table")
print result.fetchall()
```

When SQLAlchemy 0.2 was introduced the `Connection` object was added, providing the ability to explicitly maintain the scope of the DBAPI connection:

```
conn = engine.connect()
result = conn.execute("select * from table")
print result.fetchall()
conn.close()
```

The result returned by the `execute` method of `Engine` or `Connection` is called a `ResultProxy`, which offers an interface similar to the DBAPI cursor but with richer behavior. The `Engine`, `Connection`, and `ResultProxy` correspond to the DBAPI module, an instance of a specific DBAPI connection, and an instance of a specific DBAPI cursor, respectively.

Behind the scenes, the `Engine` references an object called a `Dialect`. The `Dialect` is an abstract class for which many implementations exist, each one targeted at a specific DBAPI/database combination. A `Connection` created on behalf of the `Engine` will refer to this `Dialect` for all decisions, which may have varied behaviors depending on the target DBAPI and database in use.

The `Connection`, when created, will procure and maintain an actual DBAPI connection from a repository known as a `Pool` that's also associated with the `Engine`. The `Pool` is responsible for creating new DBAPI connections and, usually, maintaining them in an in-memory pool for frequent re-use.

During a statement execution, an additional object called an `ExecutionContext` is created by the `Connection`. The object lasts from the point of execution throughout the lifespan of the `ResultProxy`. It may also be available as a specific subclass for some DBAPI/database combinations.

Figure 20.2 illustrates all of these objects and their relationships to each other as well as to the DBAPI components.

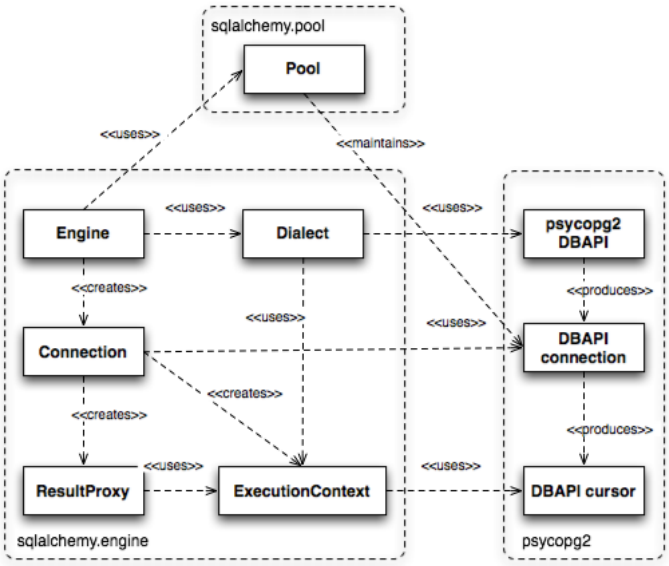

Figure 20.2: Engine, Connection, ResultProxy API

Dealing with DBAPI Variability

For the task of managing variability in DBAPI behavior, first we'll consider the scope of the problem. The DBAPI specification, currently at version two, is written as a series of API definitions which allow for a wide degree of variability in behavior, and leave a good number of areas undefined. As a result, real-life DBAPIs exhibit a great degree of variability in several areas, including when Python unicode strings are acceptable and when they are not; how the "last inserted id"—that is, an autogenerated primary key—may be acquired after an INSERT statement; and how bound parameter values may be specified and interpreted. They also have a large number of idiosyncratic type-oriented behaviors, including the handling of binary, precision numeric, date, Boolean, and unicode data.

SQLAlchemy approaches this by allowing variability in both `Dialect` and `ExecutionContext` via multi-level subclassing. Figure 20.3 illustrates the relationship between `Dialect` and `ExecutionContext` when used with the psycopg2 dialect. The `PGDialect` class provides behaviors that are specific to the usage of the PostgreSQL database, such as the ARRAY datatype and schema catalogs; the `PGDialect_psycopg2` class then provides behaviors specific to the psycopg2 DBAPI, including unicode data handlers and server-side cursor behavior.

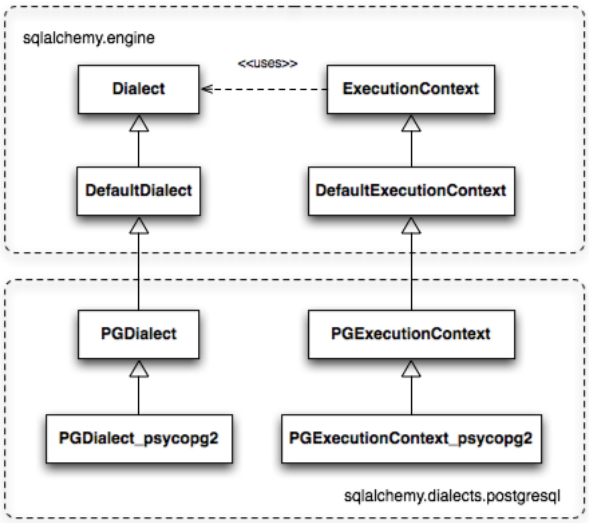

Figure 20.3: Simple Dialect/ExecutionContext hierarchy

A variant on the above pattern presents itself when dealing with a DBAPI that supports multiple databases. Examples of this include pyodbc, which deals with any number of database backends via ODBC, and zxjdbc, a Jython-only driver which deals with JDBC. The above relationship is augmented by the use of a mixin class from the `sqlalchemy.connectors` package which provides DBAPI behavior that is common to multiple backends. Figure 20.4 illustrates the common functionality of `sqlalchemy.connectors.pyodbc` shared among pyodbc-specific dialects for MySQL and Microsoft SQL Server.

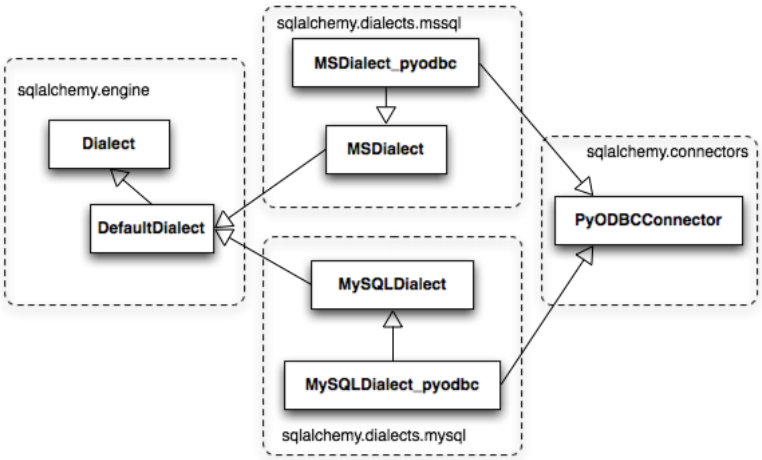

Figure 20.4: Common DBAPI behavior shared among dialect hierarchies

The `Dialect` and `ExecutionContext` objects provide a means to define every interaction with the database and DBAPI, including how connection arguments are formatted and how special quirks during statement execution are handled. The `Dialect` is also a factory for SQL compilation constructs that render SQL correctly for the target database, and type objects which define how Python data should be marshaled to and from the target DBAPI and database.

20.4. Schema Definition

With database connectivity and interactivity established, the next task is to provide for the creation and manipulation of backend-agnostic SQL statements. To achieve this, we need to define first how we will refer to the tables and columns present in a database—the so-called "schema". Tables and columns represent how data is organized, and most SQL statements consist of expressions and commands referring to these structures.

An ORM or data access layer needs to provide programmatic access to the SQL language; at the base is a programmatic system of describing tables and columns. This is where SQLAlchemy offers the first strong division of Core and ORM, by offering the `Table` and `Column` constructs that describe the structure of the database independently of a user's model class definition. The rationale behind the division of schema definition from object relational mapping is that the relational schema can be designed unambiguously in terms of the relational database, including platform-specific details if necessary, without being muddled by object-relational concepts—these remain a separate concern. Being independent of the ORM component also means the schema description system is just as useful for any other kind of object-relational system which may be built on the Core.

The `Table` and `Column` model falls under the scope of what's referred to as *metadata*, offering a collection object called `MetaData` to represent a collection of `Table` objects. The structure is derived mostly from Martin Fowler's description of "Metadata Mapping" in *Patterns of Enterprise Application Architecture*. Figure 20.5 illustrates some key elements of the `sqlalchemy.schema` package.

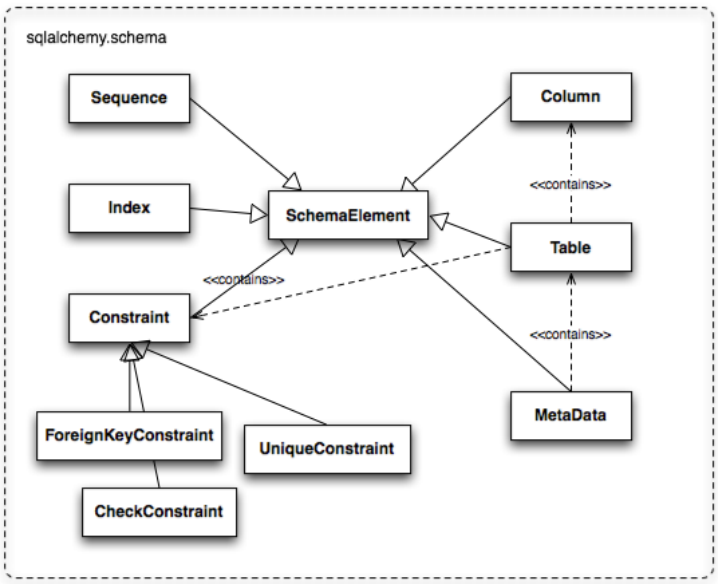

Figure 20.5: Basic sqlalchemy.schema objects

`Table` represents the name and other attributes of an actual table present in a target schema. Its collection of `Column` objects represents naming and typing information about individual table columns. A full array of objects describing constraints, indexes, and sequences is provided to fill in many more details, some of which impact the behavior of the engine and SQL construction system. In particular, `ForeignKeyConstraint` is central to determining how two tables should be joined.

`Table` and `Column` in the schema package are unique versus the rest of the package in that they are dual-inheriting, both from the `sqlalchemy.schema` package and the `sqlalchemy.sql.expression` package, serving not just as schema-level constructs, but also as core syntactical units in the SQL expression language. This relationship is illustrated in Figure 20.6.

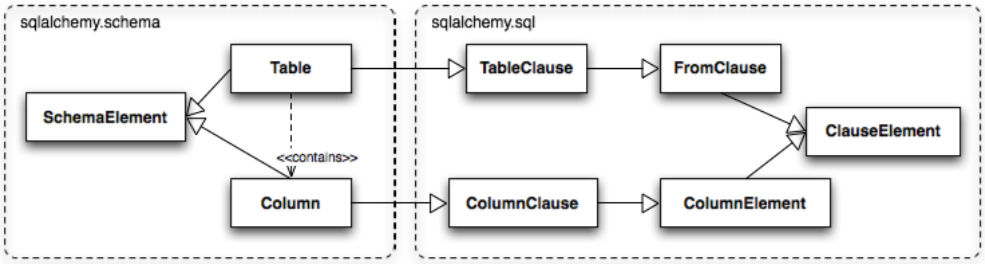

Figure 20.6: The dual lives of Table and Column

In Figure 20.6 we can see that `Table` and `Column` inherit from the SQL world as specific forms of "things you can select from", known as a `FromClause`, and "things you can use in a SQL expression", known as a `ColumnElement`.

20.5. SQL Expressions

During SQLAlchemy's creation, the approach to SQL generation wasn't clear. A textual language might have been a likely candidate; this is a common approach which is at the core of well-known object-relational tools like Hibernate's HQL. For Python, however, a more intriguing choice was available: using Python objects and expressions to generatively construct expression tree structures, even re-purposing Python operators so that operators could be given SQL statement behavior.

While it may not have been the first tool to do so, full credit goes to the SQLBuilder library included in Ian Bicking's SQLObject as the inspiration for the system of Python objects and operators used by SQLAlchemy's expression language. In this approach, Python objects represent lexical portions of a SQL expression. Methods on those objects, as well as overloaded operators, generate new lexical constructs derived from them. The most common object is the "Column" object—SQLObject would represent these on an ORM-mapped class using a namespace accessed via the `.q` attribute; SQLAlchemy named the attribute `.c`. The `.c` attribute remains today on Core selectable elements, such as those representing tables and select statements.

Expression Trees

A SQLAlchemy SQL expression construct is very much the kind of structure you'd create if you were parsing a SQL statement—it's a parse tree, except the developer creates the parse tree directly, rather than deriving it from a string. The core type of node in this parse tree is called `ClauseElement`, and Figure 20.7 illustrates the relationship of `ClauseElement` to some key classes.

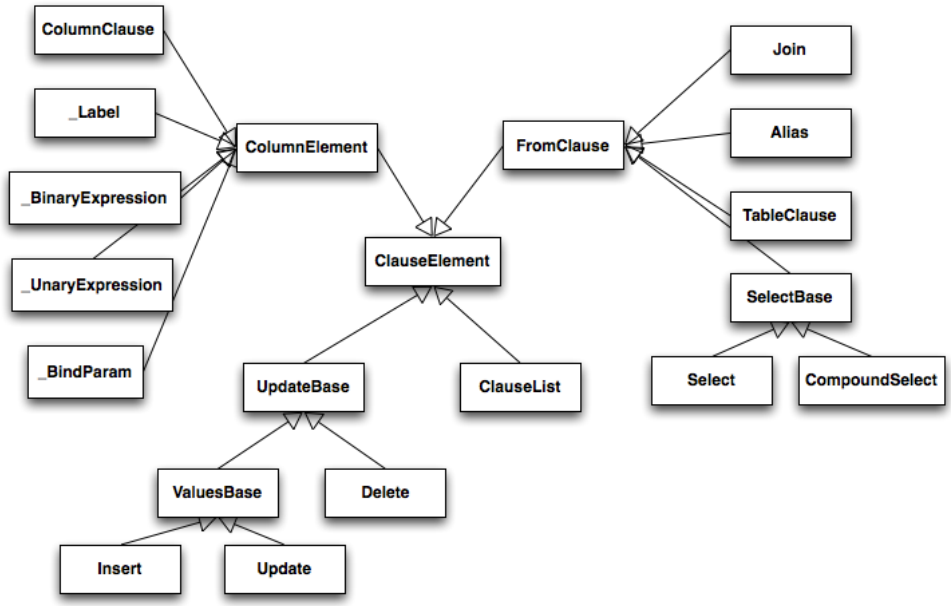

Figure 20.7: Basic expression hierarchy

Through the use of constructor functions, methods, and overloaded Python operator functions, a structure for a statement like:

```
SELECT id FROM user WHERE name = ?
```

might be constructed in Python like:

```
from sqlalchemy.sql import table, column, select
user = table('user', column('id'), column('name'))
stmt = select([user.c.id]).where(user.c.name=='ed')
```

The structure of the above `select` construct is shown in Figure 20.8. Note the representation of the literal value `'ed'` is contained within the `_BindParam` construct, thus causing it to be rendered as a bound parameter marker in the SQL string using a question mark.

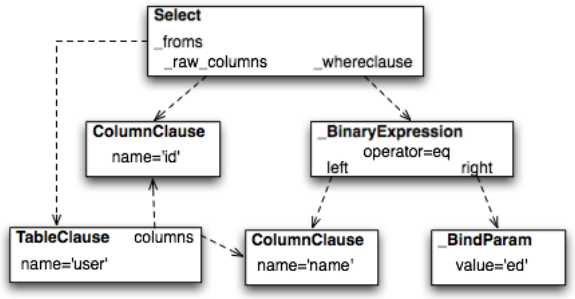

Figure 20.8: Example expression tree

From the tree diagram, one can see that a simple descending traversal through the nodes can quickly create a rendered SQL statement, as we'll see in greater detail in the section on statement compilation.

Python Operator Approach

In SQLAlchemy, an expression like this:

```
column('a') == 2
```

produces neither `True` nor `False`, but instead a SQL expression construct. The key to this is to overload operators using the Python special operator functions: e.g., methods like `__eq__`, `__ne__`, `__le__`, `__lt__`, `__add__`, `__mul__`. Column-oriented expression nodes provide overloaded Python operator behavior through the usage of a mixin called `ColumnOperators`. Using operator overloading, an expression `column('a') == 2` is equivalent to:

```
from sqlalchemy.sql.expression import _BinaryExpression
from sqlalchemy.sql import column, bindparam
from sqlalchemy.operators import eq

_BinaryExpression(
    left=column('a'),
    right=bindparam('a', value=2, unique=True),
    operator=eq
)
```

The `eq` construct is actually a function originating from the Python `operator` built-in. Representing operators as an object (i.e., `operator.eq`) rather than a string (i.e., `=`) allows the string representation to be defined at statement compilation time, when database dialect information is known.

Compilation

The central class responsible for rendering SQL expression trees into textual SQL is the `Compiled` class. This class has two primary subclasses, `SQLCompiler` and `DDLCompiler`. `SQLCompiler` handles SQL rendering operations for SELECT, INSERT, UPDATE, and DELETE statements, collectively classified as DQL (data query language) and DML (data manipulation language), while `DDLCompiler` handles various CREATE and DROP statements, classified as DDL (data definition language). There is an additional class hierarchy focused around string representations of types, starting at `TypeCompiler`. Individual dialects then provide their own subclasses of all three compiler types to define SQL language aspects specific to the target database. Figure 20.9 provides an overview of this class hierarchy with respect to the PostgreSQL dialect.

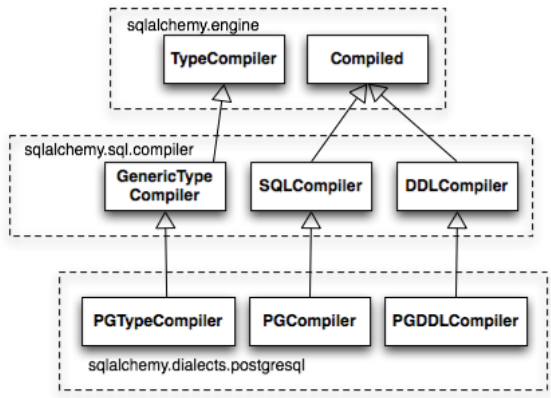

Figure 20.9: Compiler hierarchy, including PostgreSQL-specific implementation

The `Compiled` subclasses define a series of *visit* methods, each one referred to by a particular subclass of `ClauseElement`. A hierarchy of `ClauseElement` nodes is walked and a statement is constructed by recursively concatenating the string output of each visit function. As this proceeds, the `Compiled` object maintains state regarding anonymous identifier names, bound parameter names, and nesting of subqueries, among other things, all of which aim for the production of a string SQL statement as well as a final collection of bound parameters with default values. Figure 20.10 illustrates the process of visit methods resulting in textual units.

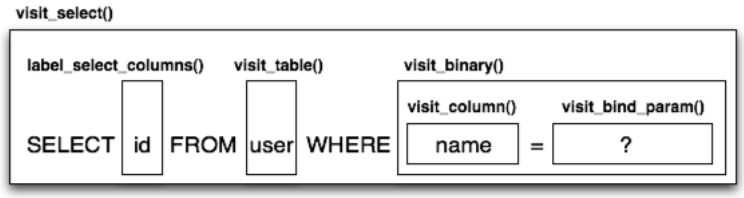

Figure 20.10: Call hierarchy of a statement compilation

A completed `Compiled` structure contains the full SQL string and collection of bound values. These are coerced by an `ExecutionContext` into the format expected by the DBAPI's `execute` method, which includes such considerations as the treatment of a unicode statement object, the type of collection used to store bound values, as well as specifics on how the bound values themselves should be coerced into representations appropriate to the DBAPI and target database.

20.6. Class Mapping with the ORM

We now shift our attention to the ORM. The first goal is to use the system of table metadata we've defined to allow mapping of a user-defined class to a collection of columns in a database table. The second goal is to allow the definition of relationships between user-defined classes, based on relationships between tables in a database.

SQLAlchemy refers to this as "mapping", following the well known Data Mapper pattern described in Fowler's *Patterns of Enterprise Architecture*. Overall, the SQLAlchemy ORM draws heavily from the practices detailed by Fowler. It's also heavily influenced by the famous Java relational mapper Hibernate and Ian Bicking's SQLObject product for Python.

Classical vs. Declarative

We use the term *classical mapping* to refer to SQLAlchemy's system of applying an object-relational data mapping to an existing user class. This form considers the `Table` object and the user-defined class to be two individually defined entities which are joined together via a function called `mapper`. Once `mapper` has been applied to a user-defined class, the class takes on new attributes that correspond to columns in the table:

```
class User(object):
    pass

mapper(User, user_table)

# now User has an ".id" attribute
User.id
```

`mapper` can also affix other kinds of attributes to the class, including attributes which correspond to references to other kinds of objects, as well as arbitrary SQL expressions. The process of affixing arbitrary attributes to a class is known in the Python world as "monkeypatching"; however, since we are doing it in a data-driven and non-arbitrary way, the spirit of the operation is better expressed with the term *class instrumentation*.

Modern usage of SQLAlchemy centers around the Declarative extension, which is a configurational system that resembles the common active-record-like class declaration system used by many other object-relational tools. In this system, the end user explicitly defines attributes inline with the class definition, each representing an attribute on the class that is to be mapped. The `Table` object, in most cases, is not mentioned explicitly, nor is the `mapper` function; only the class, the `Column` objects, and other ORM-related attributes are named:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
```

It may appear, above, that the class instrumentation is being achieved directly by our placement of `id = Column()`, but this is not the case. The Declarative extension uses a Python metaclass, which is a handy way to run a series of operations each time a new class is first declared, to generate a new `Table` object from what's been declared, and to pass it to the `mapper` function along with the class. The `mapper` function then does its job in exactly the same way, patching its own attributes onto the class, in this case towards the `id` attribute, and replacing what was there previously. By the time the metaclass initialization is complete (that is, when the flow of execution leaves the block delineated by `User`), the `Column` object marked by `id` has been moved into a new `Table`, and `User.id` has been replaced by a new attribute specific to the mapping.

It was always intended that SQLAlchemy would have a shorthand, declarative form of configuration. However, the creation of Declarative was delayed in favor of continued work solidifying the mechanics of classical mapping. An interim extension called ActiveMapper, which later became the Elixir project, existed early on. It redefines mapping constructs in a higher-level declaration system. Declarative's goal was to reverse the direction of Elixir's heavily abstracted approach by establishing a system that preserved SQLAlchemy classical mapping concepts almost exactly, only reorganizing how they are used to be less verbose and more amenable to class-level extensions than a classical mapping would be.

Whether classical or declarative mapping is used, a mapped class takes on new behaviors that allow it to express SQL constructs in terms of its attributes. SQLAlchemy originally followed SQLObject's behavior of using a special attribute as the source of SQL column expressions, referred to by SQLAlchemy as `.c`, as in this example:

```
result = session.query(User).filter(User.c.username == 'ed').all()
```

In version 0.4, however, SQLAlchemy moved the functionality into the mapped attributes themselves:

```
result = session.query(User).filter(User.username == 'ed').all()
```

This change in attribute access proved to be a great improvement, as it allowed the column-like objects present on the class to gain additional class-specific capabilities not present on those originating directly from the underlying `Table` object. It also allowed usage integration between different kinds of class attributes, such as attributes which refer to table columns directly, attributes that refer to SQL expressions derived from those columns, and attributes that refer to a related class. Finally, it provided a symmetry between a mapped class, and an instance of that mapped class, in that the same attribute could take on different behavior depending on the type of parent. Class-bound attributes return SQL expressions while instance-bound attributes return actual data.

Anatomy of a Mapping

The `id` attribute that's been attached to our `User` class is a type of object known in Python as a *descriptor*, an object that has `__get__`, `__set__`, and `__del__` methods, which the Python runtime defers to for all class and instance operations involving this attribute. SQLAlchemy's implementation is known as an `InstrumentedAttribute`, and we'll illustrate the world behind this facade with another example. Starting with a `Table` and a user defined class, we set up a mapping that has just one mapped column, as well as a `relationship`, which defines a reference to a related class:

```
user_table = Table("user", metadata,
    Column('id', Integer, primary_key=True),
)

class User(object):
    pass

mapper(User, user_table, properties={
    'related':relationship(Address)
})
```

When the mapping is complete, the structure of objects related to the class is detailed in Figure 20.11.

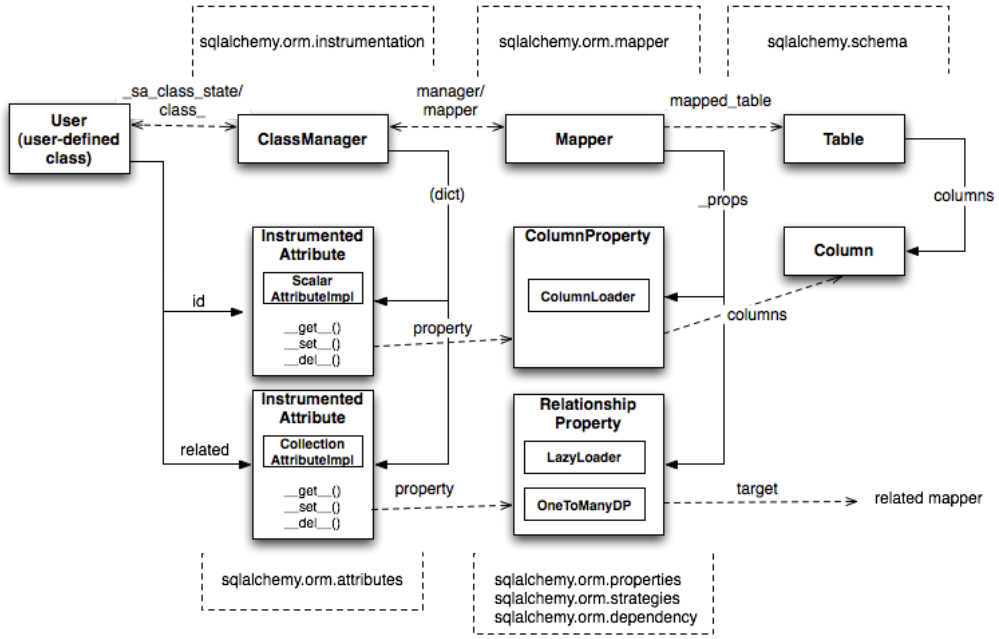

Figure 20.11: Anatomy of a mapping

The figure illustrates a SQLAlchemy mapping defined as two separate layers of interaction between the user-defined class and the table metadata to which it is mapped. Class instrumentation is pictured towards the left, while SQL and database functionality is pictured towards the right. The general pattern at play is that object composition is used to isolate behavioral roles, and object inheritance is used to distinguish amongst behavioral variances within a particular role.

Within the realm of class instrumentation, the `ClassManager` is linked to the mapped class, while its collection of `InstrumentedAttribute` objects are linked to each attribute mapped on the class. `InstrumentedAttribute` is also the public-facing Python descriptor mentioned previously, and produces SQL expressions when used in a class-based expression (e.g., `User.id==5`). When dealing with an instance of `User`, `InstrumentedAttribute` delegates the behavior of the attribute to an `AttributeImpl` object, which is one of several varieties tailored towards the type of data being represented.

Towards the mapping side, the `Mapper` represents the linkage of a user-defined class and a selectable unit, most typically `Table`. `Mapper` maintains a collection of per-attribute objects known as `MapperProperty`, which deals with the SQL representation of a particular attribute. The most common variants of `MapperProperty` are `ColumnProperty`, representing a mapped column or SQL expression, and `RelationshipProperty`, representing a linkage to another mapper.

`MapperProperty` delegates attribute loading behavior—including how the attribute renders in a SQL statement and how it is populated from a result row—to a `LoaderStrategy` object, of which there are several varieties. Different `LoaderStrategies` determine if the loading behavior of an attribute is *deferred*, *eager*, or *immediate*. A default version is chosen at mapper configuration time, with the option to use an alternate strategy at query time. `RelationshipProperty` also references a `DependencyProcessor`, which handles how inter-mapper dependencies and attribute synchronization should proceed at flush time. The choice of `DependencyProcessor` is based on the relational geometry of the *parent* and *target* selectables linked to the relationship.

The `Mapper` / `RelationshipProperty` structure forms a graph, where `Mapper` objects are nodes and `RelationshipProperty` objects are directed edges. Once the full set of mappers have been declared by an application, a deferred "initialization" step known as the *configuration* proceeds. It is used mainly by each `RelationshipProperty` to solidify the details between its *parent* and *target* mappers, including choice of `AttributeImpl` as well as `DependencyProcessor`. This graph is a key data structure used throughout the operation of the ORM. It participates in operations such as the so-called "cascade" behavior that defines how operations should propagate along object paths, in query operations where related objects and collections are "eagerly" loaded at once, as well as on the object flushing side where a dependency graph of all objects is established before firing off a series of persistence steps.

20.7. Query and Loading Behavior

SQLAlchemy initiates all object loading behavior via an object called `Query`. The basic state `Query` starts with includes the *entities*, which is the list of mapped classes and/or individual SQL expressions to be queried. It also has a reference to the `Session`, which represents connectivity to one or more databases, as well as a cache of data that's been accumulated with respect to transactions on those connections. Below is a rudimentary usage example:

```
from sqlalchemy.orm import Session
session = Session(engine)
query = session.query(User)
```

We create a `Query` that will yield instances of `User`, relative to a new `Session` we've created. `Query` provides a generative builder pattern in the same way as the `select` construct discussed previously, where additional criteria and modifiers are associated with a statement construct one method call at a time. When an iterative operation is called on the `Query`, it constructs a SQL expression construct representing a SELECT, emits it to the database, and then interprets the result set rows as ORM-oriented results corresponding to the initial set of entities being requested.

`Query` makes a hard distinction between the *SQL rendering* and the *data loading* portions of the operation. The former refers to the construction of a SELECT statement, the latter to the interpretation of SQL result rows into ORM-mapped constructs. Data loading can, in fact, proceed without a SQL rendering step, as the `Query` may be asked to interpret results from a textual query hand-composed by the user.

Both SQL rendering and data loading utilize a recursive descent through the graph formed by the series of lead `Mapper` objects, considering each column- or SQL-expression-holding `ColumnProperty` as a leaf node and each `RelationshipProperty` which is to be included in the query via a so-called "eager-load" as an edge leading to another `Mapper` node. The traversal and action to take at each node is ultimately the job of each `LoaderStrategy` associated with every `MapperProperty`, adding columns and joins to the SELECT statement being built in the SQL rendering phase, and producing Python functions that process result rows in the data loading phase.

The Python functions produced in the data loading phase each receive a database row as they are fetched, and produce a possible change in the state of a mapped attribute in memory as a result. They are produced for a particular attribute conditionally, based on examination of the first incoming row in the result set, as well as on loading options. If a load of the attribute is not to proceed, no callable function is produced.

Figure 20.12 illustrates the traversal of several `LoaderStrategy` objects in a *joined eager loading* scenario, illustrating their connection to a rendered SQL statement which occurs during the `_compile_context` method of `Query`. It also shows generation of *row population* functions which receive result rows and populate individual object attributes, a process which occurs within the `instances` method of `Query`.

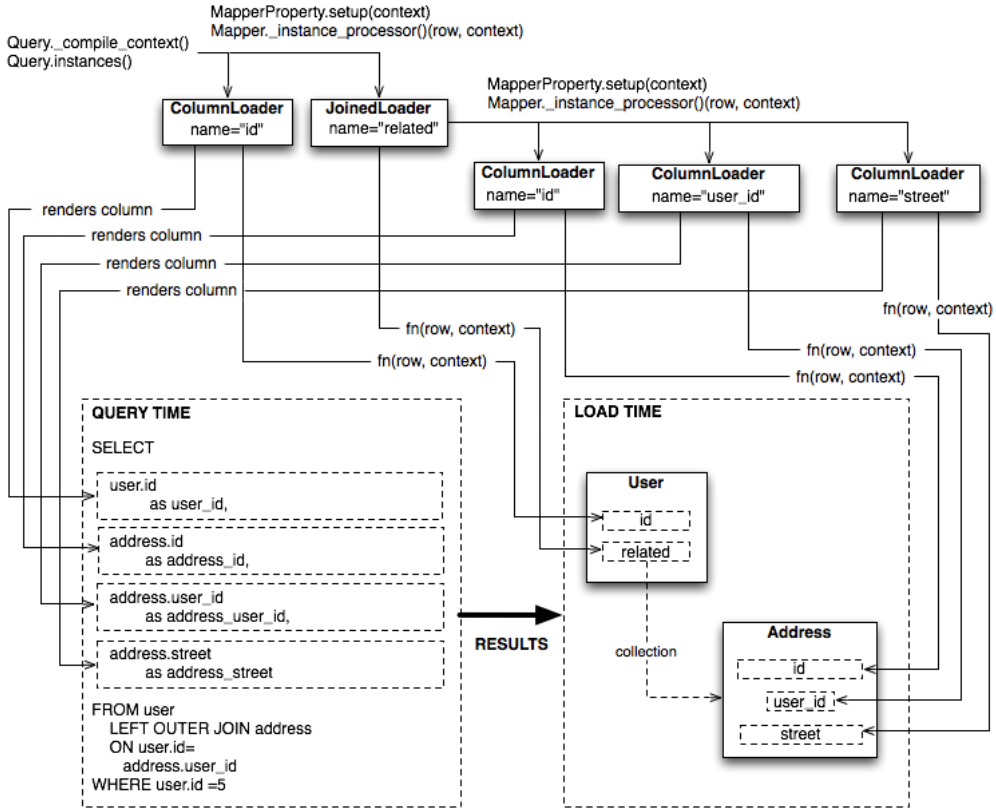

Figure 20.12: Traversal of loader strategies including a joined eager load

SQLAlchemy's early approach to populating results used a traditional traversal of fixed object methods associated with each strategy to receive each row and act accordingly. The loader callable system, first introduced in version 0.5, represented a dramatic leap in performance, as many decisions regarding row handling could be made just once up front instead of for each row, and a significant number of function calls with no net effect could be eliminated.

20.8. Session/Identity Map

In SQLAlchemy, the `Session` object presents the public interface for the actual usage of the ORM—that is, loading and persisting data. It provides the starting point for queries and persistence operations for a given database connection.

The `Session`, in addition to serving as the gateway for database connectivity, maintains an active reference to the set of all mapped entities which are present in memory relative to that `Session`. It's in this way that the `Session` implements a facade for the *identity map* and *unit of work* patterns, both identified by Fowler. The identity map maintains a database-identity-unique mapping of all objects for a particular `Session`, eliminating the problems introduced by duplicate identities. The unit of work builds on the identity map to provide a system of automating the process of persisting all changes in state to the database in the most effective manner possible. The actual persistence step is known as a "flush", and in modern SQLAlchemy this step is usually automatic.

Development History

The `Session` started out as a mostly concealed system responsible for the single task of emitting a flush. The flush process involves emitting SQL statements to the database, corresponding to changes in the state of objects tracked by the unit of work system and thereby synchronizing the current state of the database with what's in memory. The flush has always been one of the most complex operations performed by SQLAlchemy.

The invocation of *flush* started out in very early versions behind a method called `commit`, and it was a method present on an implicit, thread-local object called `objectstore`. When one used SQLAlchemy 0.1, there was no need to call `Session.add`, nor was there any concept of an explicit `Session` at all. The only user-facing steps were to create mappers, create new objects, modify existing objects loaded through queries (where the queries themselves were invoked directly from each `Mapper` object), and then persist all changes via the `objectstore.commit` command. The pool of objects for a set of operations was unconditionally module-global and unconditionally thread-local.

The `objectstore.commit` model was an immediate hit with the first group of users, but the rigidity of this model quickly ran into a wall. Users new to modern SQLAlchemy sometimes lament the need to define a factory, and possibly a registry, for `Session` objects, as well as the need to keep their objects organized into just one `Session` at a time, but this is far preferable to the early days when the entire system was completely implicit. The convenience of the 0.1 usage pattern is still largely present in modern SQLAlchemy, which features a session registry normally configured to use thread local scoping.

The `Session` itself was only introduced in version 0.2 of SQLAlchemy, modeled loosely after the `Session` object present in Hibernate. This version featured integrated transactional control, where the `Session` could be placed into a transaction via the `begin` method, and completed via the `commit` method. The `objectstore.commit` method was renamed to `objectstore.flush`, and new `Session` objects could be created at any time. The `Session` itself was broken off from another object called `UnitOfWork`, which remains as a private object responsible for executing the actual flush operation.

While the flush process started as a method explicitly invoked by the user, the 0.4 series of SQLAlchemy introduced the concept of *autoflush*, which meant that a flush was emitted immediately before each query. The advantage of autoflush is that the SQL statement emitted by a query always has access on the relational side to the exact state that is present in memory, as all changes have been sent over. Early versions of SQLAlchemy couldn't include this feature, because the most common pattern of usage was that the flush statement would also commit the changes permanently. But when autoflush was introduced, it was accompanied by another feature called the *transactional* `Session`, which provided a `Session` that would start out automatically in a transaction that remained until the user called `commit` explicitly. With the introduction of this feature, the `flush` method no longer committed the data that it flushed, and could safely be called on an automated basis. The `Session` could now provide a step-by-step synchronization between in-memory state and SQL query state by flushing as needed, with nothing permanently persisted until the explicit `commit` step. This behavior is, in fact, exactly the same in Hibernate for Java. However, SQLAlchemy embraced this style of usage based on the same behavior in the Storm ORM for Python, introduced when SQLAlchemy was in version 0.3.

Version 0.5 brought more transaction integration when *post-transaction expiration* was introduced; after each `commit` or `rollback`, by default all states within the `Session` are expired (erased), to be populated again when subsequent SQL statements re-select the data, or when the attributes on the remaining set of expired objects are accessed in the context of the new transaction. Originally, SQLAlchemy was constructed around the assumption that SELECT statements should be emitted as little as possible, unconditionally. The expire-on-commit behavior was slow in coming for this reason; however, it entirely solved the issue of the `Session` which contained stale data post-transaction with no simple way to load newer data without rebuilding the full set of objects already loaded. Early on, it seemed that this problem couldn't be reasonably solved, as it wasn't apparent when the `Session` should consider the current state to be stale, and thus produce an expensive new set of SELECT statements on the next access. However, once the `Session` moved to an always-in-a-transaction model, the point of transaction end became apparent as the natural point of data expiration, as the nature of a transaction with a high degree of isolation is that it *cannot* see new data until it's committed or rolled back anyway. Different databases and configurations, of course, have varied degrees of transaction isolation, including no transactions at all. These modes of usage are entirely acceptable with SQLAlchemy's expiration model; the developer only needs to

be aware that a lower isolation level may expose un-isolated changes within a Session if multiple Sessions share the same rows. This is not at all different from what can occur when using two database connections directly.

Session Overview

Figure 20.13 illustrates a `Session` and the primary structures it deals with.

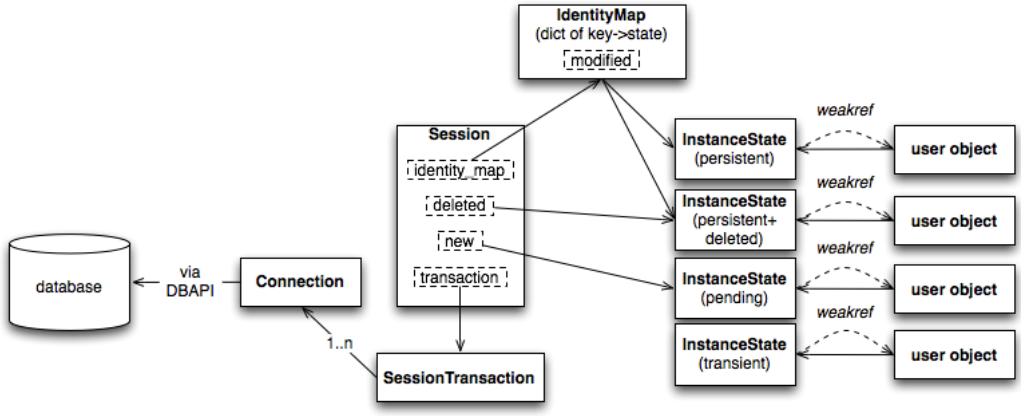

Figure 20.13: Session overview

The public-facing portions above are the `Session` itself and the collection of user objects, each of which is an instance of a mapped class. Here we see that mapped objects keep a reference to a SQLAlchemy construct called `InstanceState`, which tracks ORM state for an individual instance including pending attribute changes and attribute expiration status. `InstanceState` is the instance-level side of the attribute instrumentation discussed in the preceding section, *Anatomy of a Mapping*, corresponding to the `ClassManager` at the class level, and maintaining the state of the mapped object's dictionary (i.e., the Python `__dict__` attribute) on behalf of the `AttributeImpl` objects associated with the class.

State Tracking

The `IdentityMap` is a mapping of database identities to `InstanceState` objects, for those objects which have a database identity, which are referred to as *persistent*. The default implementation of `IdentityMap` works with `InstanceState` to self-manage its size by removing user-mapped instances once all strong references to them have been removed—in this way it works in the same way as Python's `WeakValueDictionary`. The `Session` protects the set of all objects marked as *dirty* or *deleted*, as well as pending objects marked *new*, from garbage collection, by creating strong references to those objects with pending changes. All strong references are then discarded after the flush.

`InstanceState` also performs the critical task of maintaining "what's changed" for the attributes of a particular object, using a move-on-change system that stores the "previous" value of a particular attribute in a dictionary called `committed_state` before assigning the incoming value to the object's current dictionary. At flush time, the contents of `committed_state` and the `__dict__` associated with the object are compared to produce the set of net changes on each object.

In the case of collections, a separate `collections` package coordinates with the `InstrumentedAttribute` / `InstanceState` system to maintain a collection of net changes to a particular mapped collection of objects. Common Python classes such as `set`, `list` and `dict` are subclassed before use and augmented with history-tracking mutator methods. The collection system was reworked in 0.4 to be open ended and usable for any collection-like object.

Transactional Control

`Session`, in its default state of usage, maintains an open transaction for all operations which is completed when `commit` or `rollback` is called. The `SessionTransaction` maintains a set of zero or more `Connection` objects, each representing an open transaction on a particular database. `SessionTransaction` is a lazy-initializing object that begins with no database state present. As a particular backend is required to participate in a statement execution, a `Connection` corresponding to that database is added to `SessionTransaction`'s list of connections. While a single connection at a time is common, the multiple connection scenario is supported where the specific connection used for a particular operation is determined based on configurations associated with the `Table`, `Mapper`, or SQL construct itself involved in the operation. Multiple connections can also coordinate the transaction using two-phase behavior, for those DBAPIs which provide it.

20.9. Unit of Work

The `flush` method provided by `Session` turns over its work to a separate module called `unitofwork`. As mentioned earlier, the flush process is probably the most complex function of SQLAlchemy.

The job of the unit of work is to move all of the *pending* state present in a particular `Session` out to the database, emptying out the `new`, `dirty`, and `deleted` collections maintained by the `Session`. Once completed, the in-memory state of the `Session` and what's present in the current transaction match. The primary challenge is to determine the correct series of persistence steps, and then to perform them in the correct order. This includes determining the list of INSERT, UPDATE, and DELETE statements, including those resulting from the cascade of a related row being deleted or otherwise moved; ensuring that UPDATE statements contain only those columns which were actually modified; establishing "synchronization" operations that will copy the state of primary key columns over to referencing foreign key columns, at the point at which newly generated primary key identifiers are available; ensuring that INSERTs occur in the order in which objects were added to the `Session` and as efficiently as possible; and ensuring that UPDATE and DELETE statements occur within a deterministic ordering so as to reduce the chance of deadlocks.

History

The unit of work implementation began as a tangled system of structures that was written in an ad hoc way; its development can be compared to finding the way out of a forest without a map. Early bugs and missing behaviors were solved with bolted-on fixes, and while several refactorings improved matters through version 0.5, it was not until version 0.6 that the unit of work—by that time stable, well-understood, and covered by hundreds of tests—could be rewritten entirely from scratch. After many weeks of considering a new approach that would be driven by consistent data structures, the process of rewriting it to use this new model took only a few days, as the idea was by this time well understood. It was also greatly helped by the fact that the new implementation's behavior could be carefully cross-checked against the existing version. This process shows how the first iteration of something, however awful, is still valuable as long as it provides a working model. It further shows how total rewrites of a subsystem is often not only appropriate, but an integral part of development for hard-to-develop systems.

Topological Sort

The key paradigm behind the unit of work is that of assembling the full list of actions to be taken into a data structure, with each node representing a single step; this is known in design patterns parlance as the *command pattern*. The series of "commands" within this structure is then organized into a specific ordering using a *topological sort*. A topological sort is a process that sorts items based on a *partial ordering*, that is, only certain elements must precede others. Figure 20.14 illustrates the behavior of the topological sort.

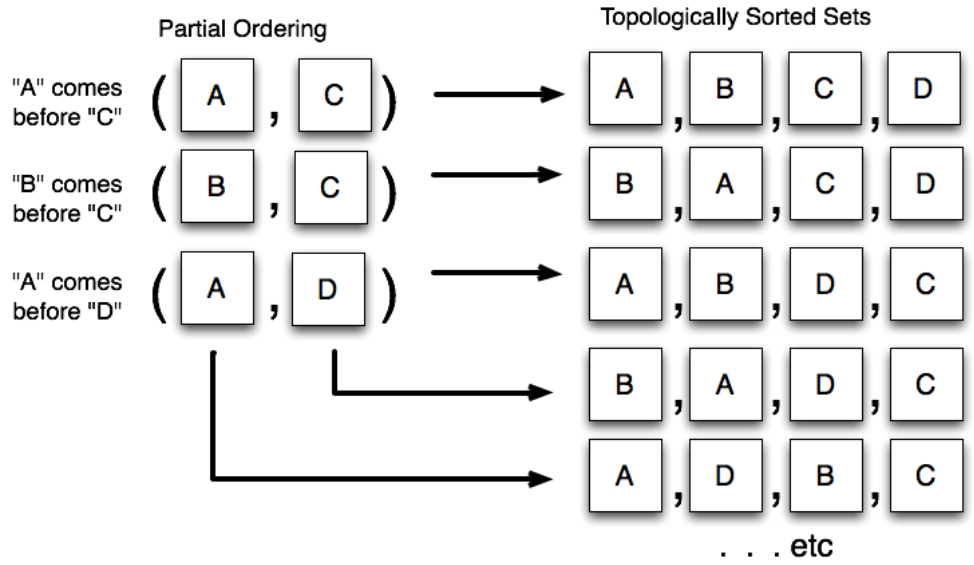

Figure 20.14: Topological sort

The unit of work constructs a partial ordering based on those persistence commands which must precede others. The commands are then topologically sorted and invoked in order. The determination of which commands precede which is derived primarily from the presence of a `relationship` that bridges two `Mapper` objects—generally, one `Mapper` is considered to be dependent on the other, as the `relationship` implies that one `Mapper` has a foreign key dependency on the other. Similar rules exist for many-to-many association tables, but here we focus on the case of one-to-many/many-to-one relationships. Foreign key dependencies are resolved in order to prevent constraint violations from occurring, with no reliance on needing to mark constraints as "deferred". But just as importantly, the ordering allows primary key identifiers, which on many platforms are only generated when an INSERT actually occurs, to be populated from a just-executed INSERT statement's result into the parameter list of a dependent row that's about to be inserted. For deletes, the same ordering is used in reverse—dependent rows are deleted before those on which they depend, as these rows cannot be present without the referent of their foreign key being present.

The unit of work features a system where the topological sort is performed at two different levels, based on the structure of dependencies present. The first level organizes persistence steps into buckets based on the dependencies between mappers, that is, full "buckets" of objects corresponding to a particular class. The second level breaks up zero or more of these "buckets" into smaller batches, to handle the case of reference cycles or self-referring tables. Figure 20.15 illustrates the "buckets" generated to insert a set of `User` objects, then a set of `Address` objects, where an intermediary step copies newly generated `User` primary key values into the `user_id` foreign key column of each `Address` object.

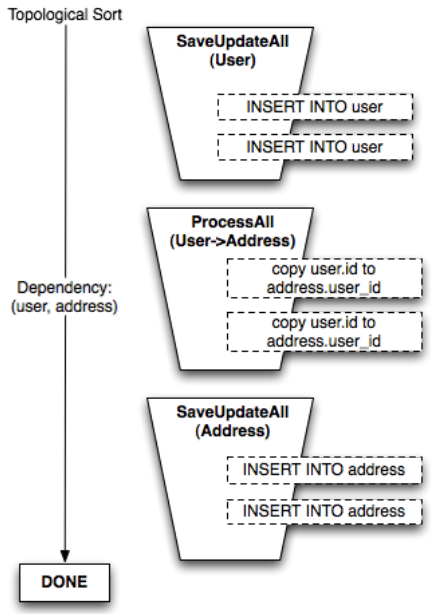

Figure 20.15: Organizing objects by mapper

In the per-mapper sorting situation, any number of `User` and `Address` objects can be flushed with no impact on the complexity of steps or how many "dependencies" must be considered.

The second level of sorting organizes persistence steps based on direct dependencies between individual objects within the scope of a single mapper. The simplest example of when this occurs is a table which contains a foreign key constraint to itself; a particular row in the table needs to be inserted before another row in the same table which refers to it. Another is when a series of tables have a *reference cycle*: table A references table B, which references table C, that then references table A. Some A objects must be inserted before others so as to allow the B and C objects to also be inserted. The table that refers to itself is a special case of reference cycle.

To determine which operations can remain in their aggregated, per-`Mapper` buckets, and which will be broken into a larger set of per-object commands, a cycle detection algorithm is applied to the set of dependencies that exist between mappers, using a modified version of a cycle detection algorithm found on Guido Van Rossum's blog. Those buckets involved in cycles are are then broken up into per-object operations and mixed into the collection of per-mapper buckets through the addition of new dependency rules from the per-object buckets back to the per-mapper buckets. Figure 20.16 illustrates the bucket of `User` objects being broken up into individual per-object commands, resulting from the addition of a new `relationship` from `User` to itself called `contact`.

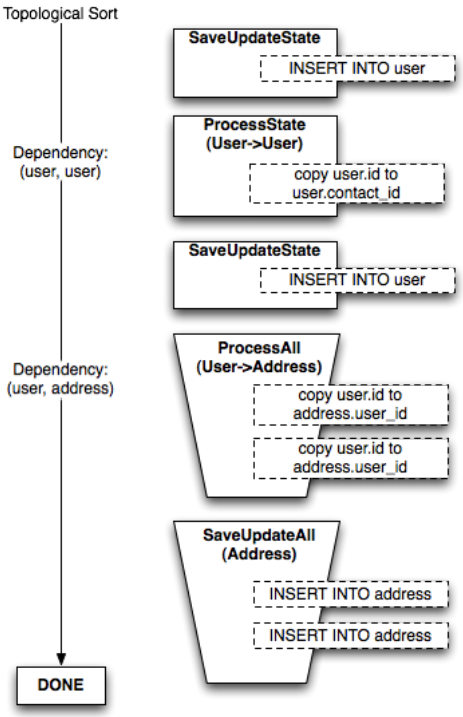

Figure 20.16: Organizing reference cycles into individual steps

The rationale behind the bucket structure is that it allows batching of common statements as much as possible, both reducing the number of steps required in Python and making possible more efficient interactions with the DBAPI, which can sometimes execute thousands of statements within a single Python method call. Only when a reference cycle exists between mappers does the more expensive per-object-dependency pattern kick in, and even then it only occurs for those portions of the object graph which require it.

20.10. Conclusion

SQLAlchemy has aimed very high since its inception, with the goal of being the most feature-rich and versatile database product possible. It has done so while maintaining its focus on relational databases, recognizing that supporting the usefulness of relational databases in a deep and comprehensive way is a major undertaking; and even now, the scope of the undertaking continues to reveal itself as larger than previously perceived.

The component-based approach is intended to extract the most value possible from each area of functionality, providing many different units that applications can use alone or in combination. This system has been challenging to create, maintain, and deliver.

The development course was intended to be slow, based on the theory that a methodical, broad-based construction of solid functionality is ultimately more valuable than fast delivery of features without foundation. It has taken a long time for SQLAlchemy to construct a consistent and well-documented user story, but throughout the process, the underlying architecture was always a step ahead, leading in some cases to the "time machine" effect where features can be added almost before users request them.

The Python language has been a reliable host (if a little finicky, particularly in the area of performance). The language's consistency and tremendously open run-time model has allowed SQLAlchemy to provide a nicer experience than that offered by similar products written in other languages.

It is the hope of the SQLAlchemy project that Python gain ever-deeper acceptance into as wide a variety of fields and industries as possible, and that the use of relational databases remains vibrant and progressive. The goal of SQLAlchemy is to demonstrate that relational databases, Python, and well-considered object models are all very much worthwhile development tools.

Back to top
Back to *The Architecture of Open Source Applications*.