SQLAlchemy Session – In Depth

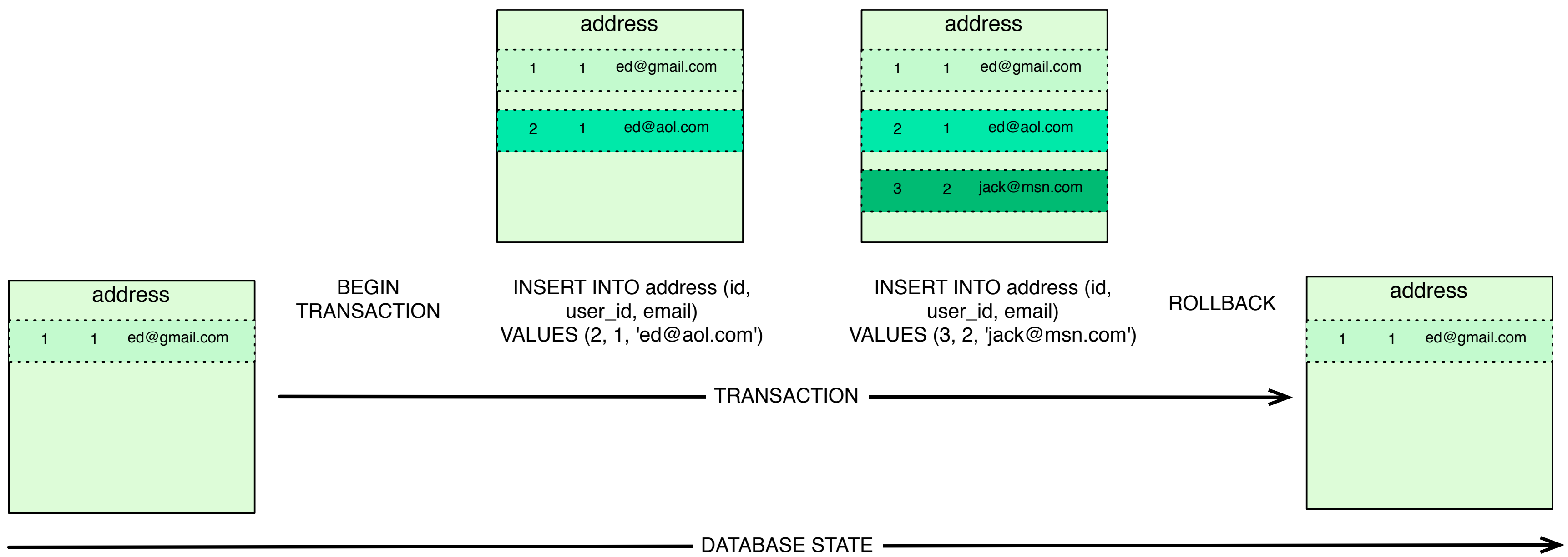
The Transaction

The Transaction

- The primary system employed by relational databases for managing data.
- Provides a scope around a series of operations with lots of desirable behaviors.
- The transaction follows the ACID model.
- Relational databases usually use transactions for all operations; if they aren't apparent, it is probably using "autocommit" by default.

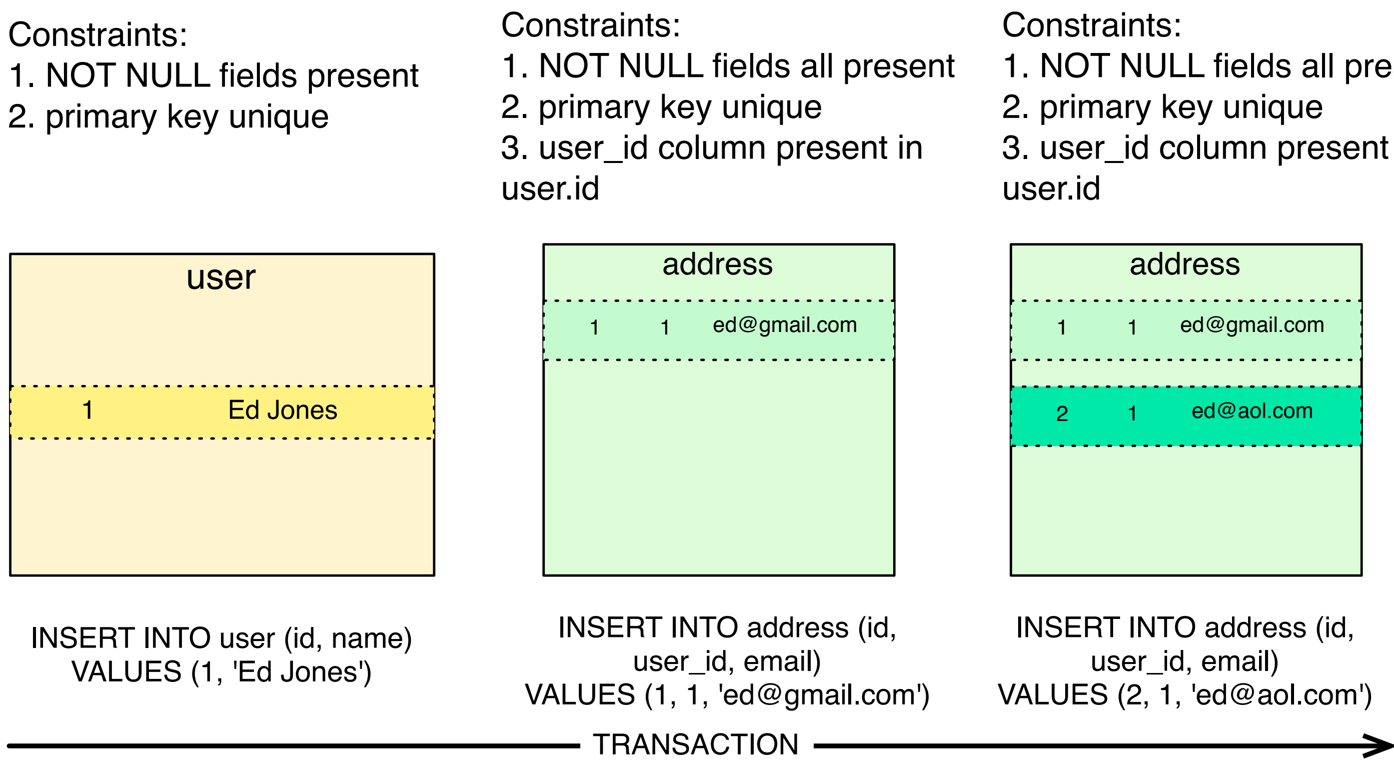
ACID Model

Transactions are **atomic** - all changes which occur can be **rolled back** to the state preceding the transaction.



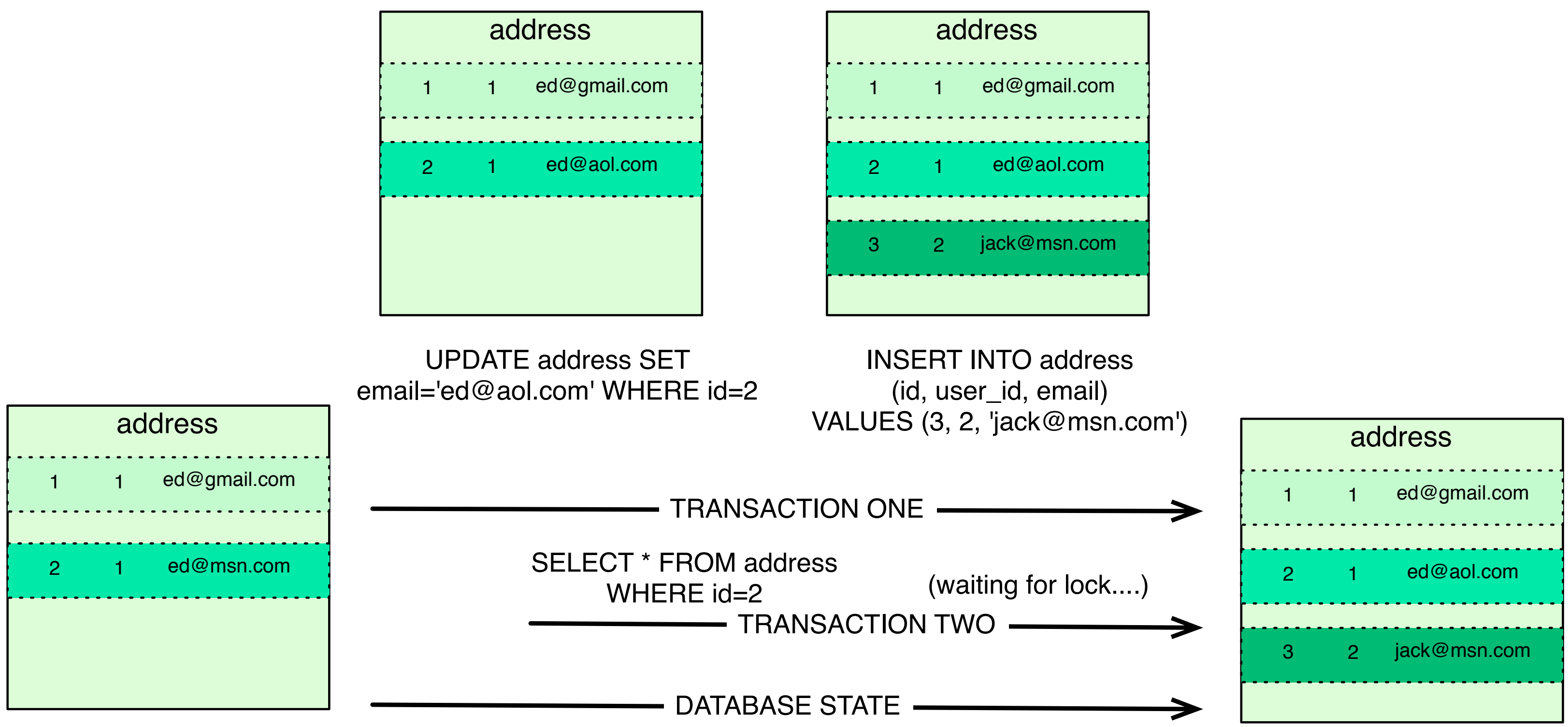
ACID Model

The transaction provides **consistency**; rules exist for how data can be created and manipulated, which often limit the order in which operations can take place



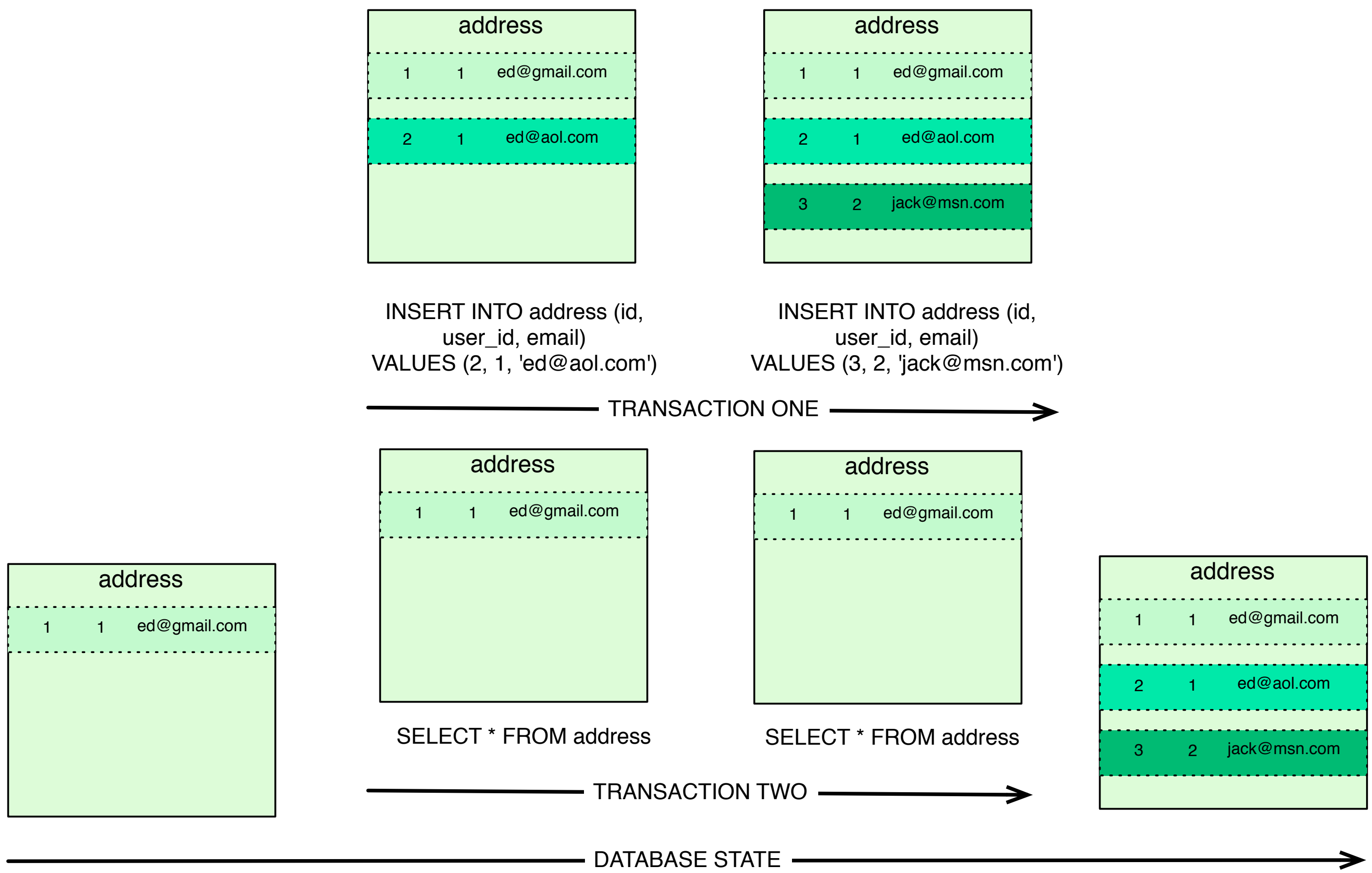
ACID Model

Transactions are **isolated** - to a varying degree, changes on the **inside** aren't visible on the **outside**, and vice versa. Historically, table and row **locks** are used to achieve this...



ACID Model

.. but most modern databases today feature **multi-version concurrency control**, which provides a high degree of isolation with much less locking



ACID Model

Transactions are **durable** - after COMMIT, you're good!

Object Relational Mappers and Transactions

Our First ORM

Configuration

```
from my_first_orm import Entity, Integer, String, \
                             Numeric, ForeignKey, relationship

class User(Entity):
    table = 'user'

    id = Integer()
    name = String()

class Address(Entity):
    table = 'address'

    id = Integer()
    user_id = ForeignKey("User.id")
    email = String()

    user = relationship("User")
```

Our First ORM

Objects are persisted using obj.save(), deleted with object.delete() – this is an **active record** style of persistence

```
user1 = User(name='Ed Jones')
user1.save()    # emits INSERT

user1.name ='Edward Jones'
user1.save()    # emits UPDATE

address1 = Address(email='ed@gmail.com', user=user1)
address1.save()    # emits INSERT

address2.delete()    # emits DELETE
```

Our First ORM

Transactions are optional, provided via implicit thread-local - else autocommit

```
from my_first_orm import Transaction

trans = Transaction.begin()

user1 = User.get(id=5)
user1.name = "Ed Jones"
user1.save()

address1 = Address(email='ed@gmail.com', user=user1)
address1.save()

trans.commit()
```

Our First ORM

Instances not coordinated on identity – "Every object for itself!"

```
>>> user1 = User.get(id=5)
>>> user2 = User.get(id=5)

>>> user1 is user2
False

>>> user1.name = 'Ed'
>>> user2.name = 'Jack'

>>> user1.name
'Ed'

>>> user2.name
'Jack'
```

Active Record Persistence

- The means of persistence is provided via the interface of each individual mapped object - object.save(), object.delete(), etc.
- Objects aren't coordinated on a particular transaction by default; "autocommit", or transaction-per-operation, is the default behavior.
- The objects don't otherwise share any connection to each other; individual queries for the same rows return different instances.
- Persist operations are immediate - an INSERT, UPDATE, or DELETE is emitted directly.

Active Record – Issues

Lack of identity coordination pushes it into save()

```
def user_process_one():
    user = User.get(id=5)
    user.name = 'Jack Jones'
    return user

def user_process_two():
    user = User.get(id=5)
    if user.name == 'Jack Jones':
        address = Address(email='jack@gmail.com', user=user)
        address.save()
    return user

user1 = user_process_one()

# order of operations here affects the outcome -
# need to save() early, possibly earlier than we'd like
user1.save()
user2 = user_process_two()

user2.save()
```

Active Record – Issues

immediate INSERT/UPDATE operations awkward, inefficient

```
for user_record in datafile:
    user = User(name=user_record.username)
    user.save()    # are all NOT NULL fields present?
                   # otherwise we can't save() it yet...

    for entry in user_record.entries:
        if entry.type == 'A':
            address = Address(user=user)
            address.email = entry.email

            # did we user.save() above? else can't do this,
            # would need to track it for later...
            address.save()

        elif entry.type == 'U':
            user.field1 = entry.field1
            user.field2 = entry.field2
            user.save()  # must we UPDATE all columns each time,
                         # and emit an UPDATE for each entry?

    # we can save() everything later, but we still must manually
    # maintain dependency ordering, and can't query as we go
```

Active Record – Issues

Instances can return stale or uncommitted data (unless they SELECT every time)

```
user1 = User.get(id=5)
user1.name = 'New Name'
user1.save()

user2 = User.get(id=5)
user2.name = 'Some Other Name'
user2.save()

# fails - user1.name still says 'New Name'
assert user1.name == 'Some Other Name'

trans = Transaction.begin()
user2.name = 'Yet Another Name'
trans.rollback()

# fails - user2.name still says 'Yet Another Name'
assert user2.name == 'Some Other Name'
```

Active Record – Issues

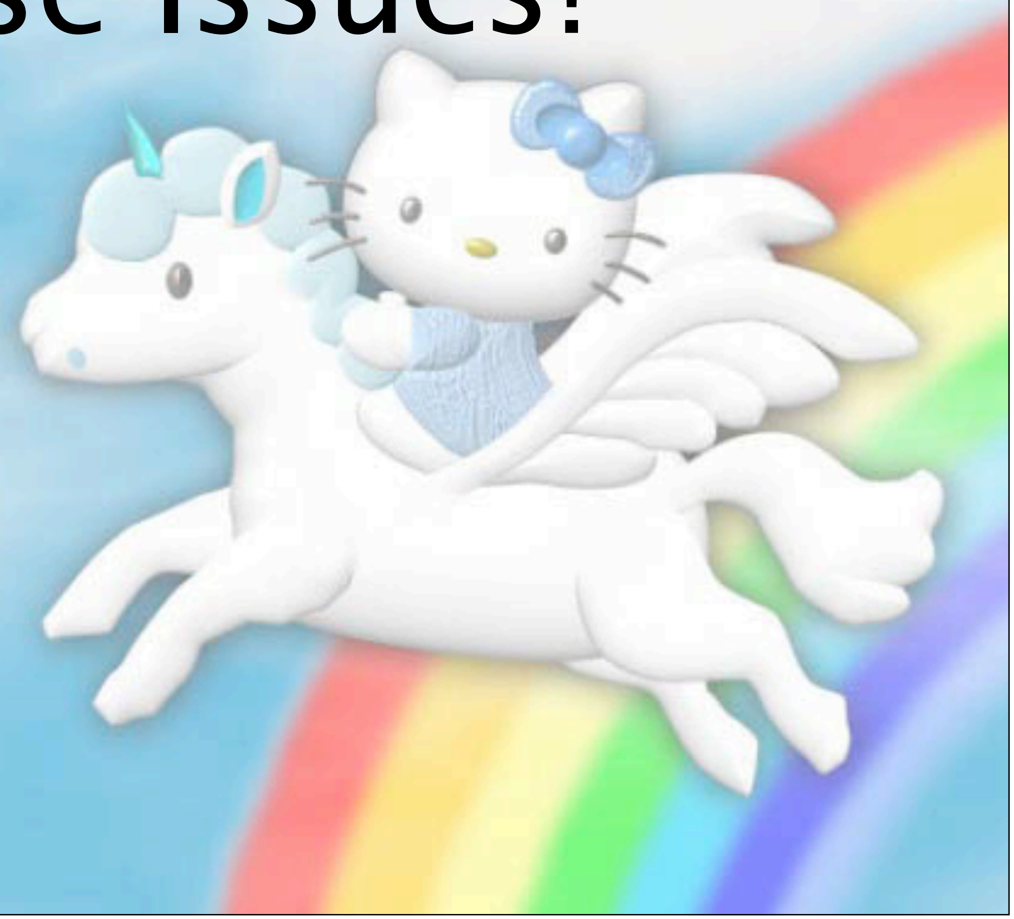
Lack of Behavioral Constraints Creates Confusion

```
queue = Queue.Queue()

def user_producer():    # thread #1: produces User objects
    trans = Transaction.begin()
    for record in data:
        user = User.get(name=record.username)
        # create User if it does not exist
        if user is None:
            user = User(name=record.username)
        user.status = record.status
        user.save()
        queue.put(user)
    trans.commit()

def user_consumer():    # thread #2: consumes User objects
    while True:
        user = queue.get()
        trans = Transaction.begin()
        if user.status == 'D':  # is this status committed or not?
             user.delete()      # is this row persisted?
                                # this code will randomly fail,
                                # either silently or loudly, based on data
        trans.commit()
        queue.task_done()
```

The Session Solves All Of These Issues!

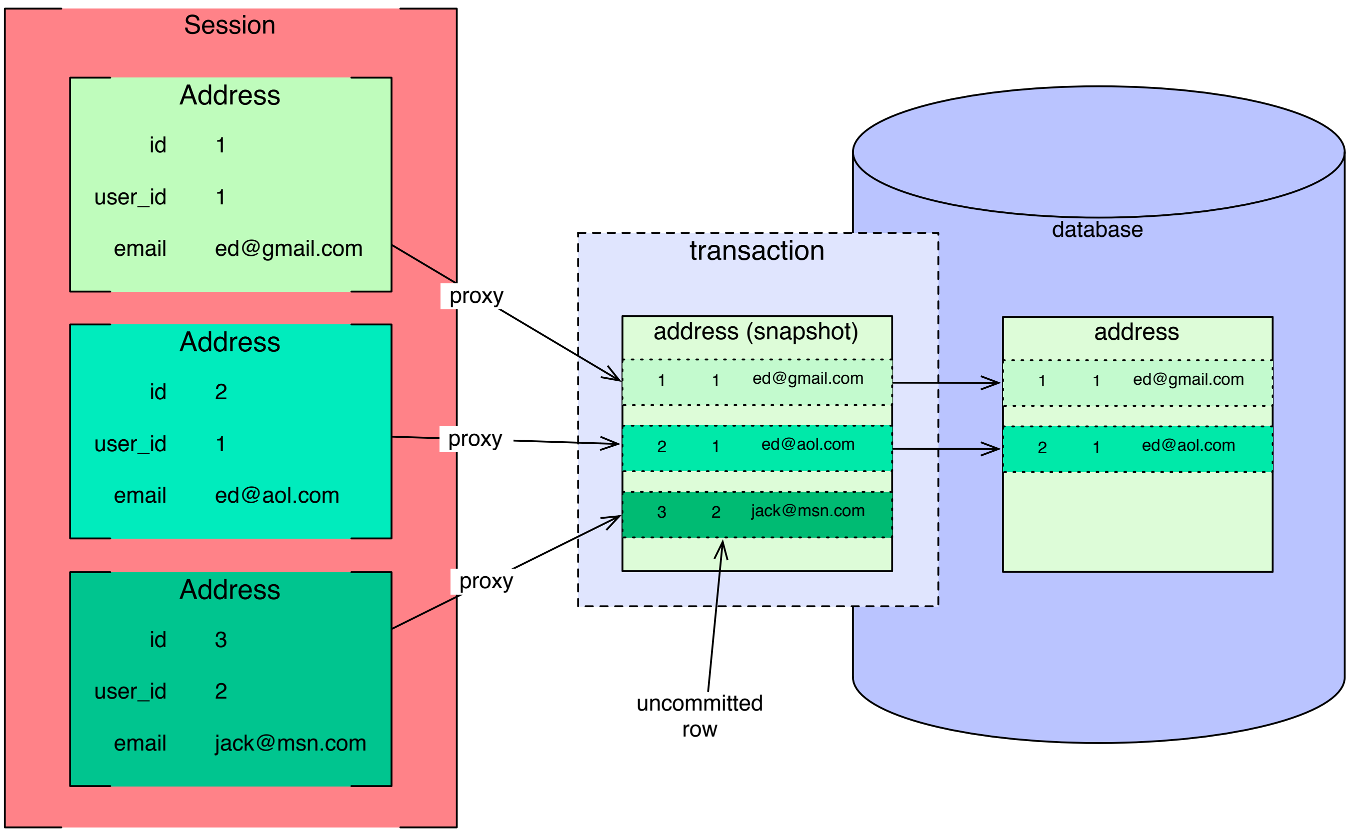


The Session Strategy

- Explicit transaction always present
- The Session maintains a cached set of transaction state, consisting of **rows.**
- A row is typically only present in the Session if it was **selected** or **inserted** in the span of that transaction.
- Objects, when associated with a Session, are **proxies** for rows, represented uniquely on **primary key identity**.
- Changes to objects are pushed out to rows before each query, and at transaction end, using **unit of work.**

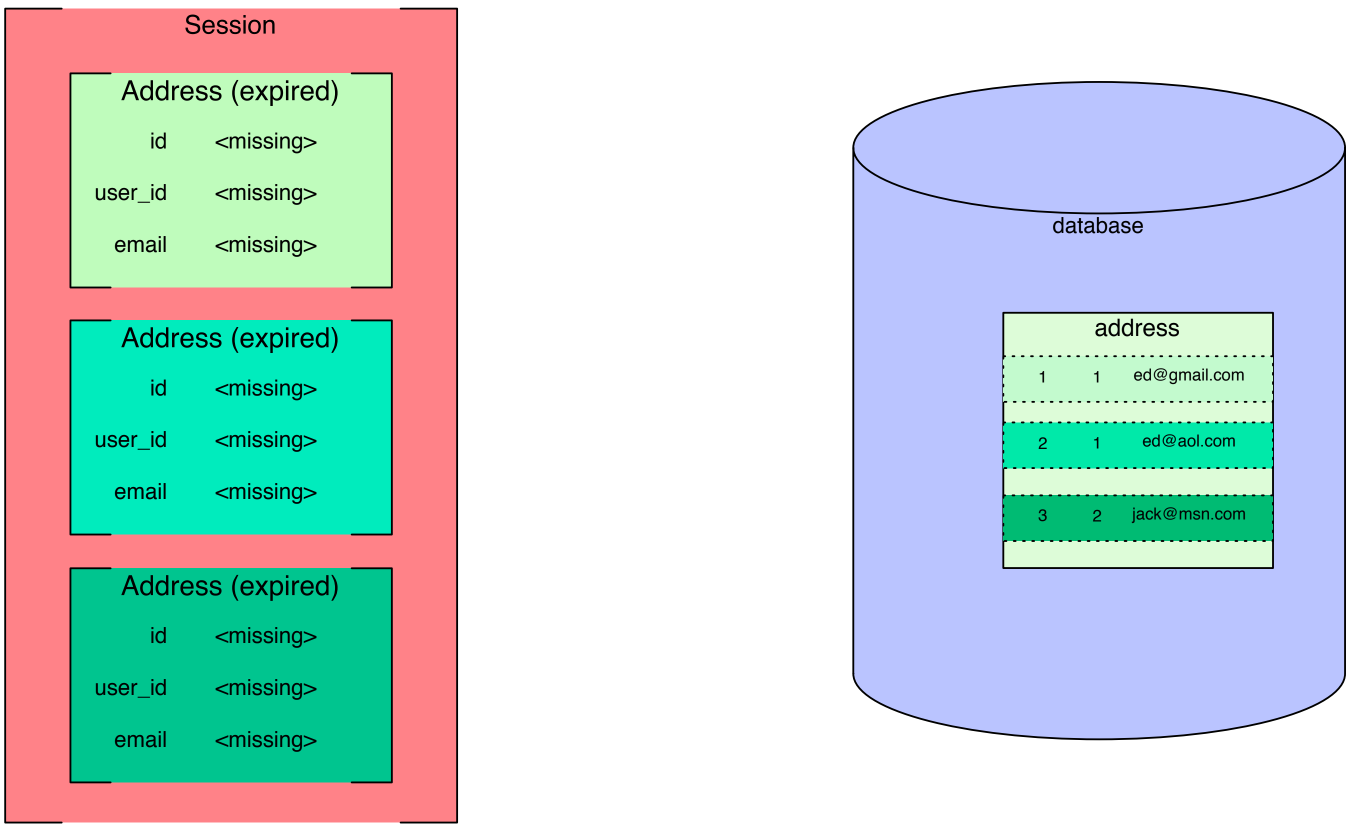
The Object as Row Proxy

An object is said to be **persistent** when it acts as a **proxy** to a row present in the transaction. This row is normally *always* known as a result of a SELECT or an INSERT.



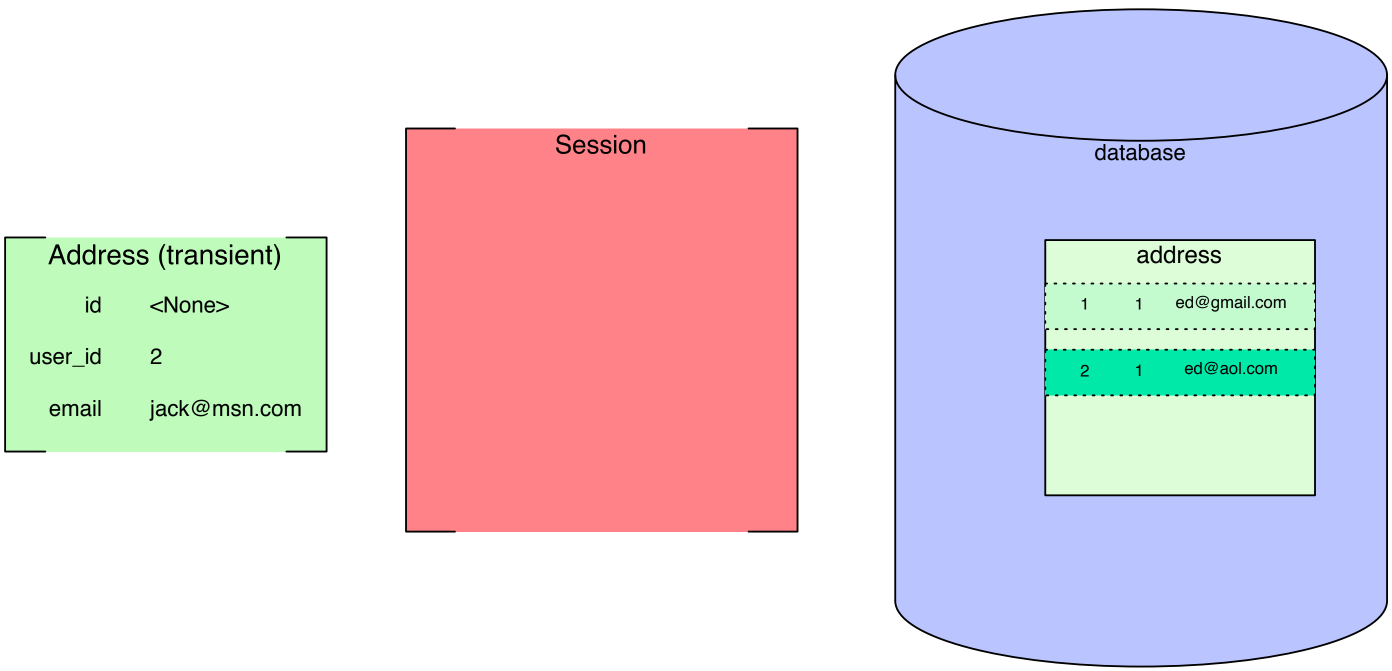
The Object as Row Proxy

With no transaction present, the state of the objects is **expired.** There is no view of the database data other than via a transaction.



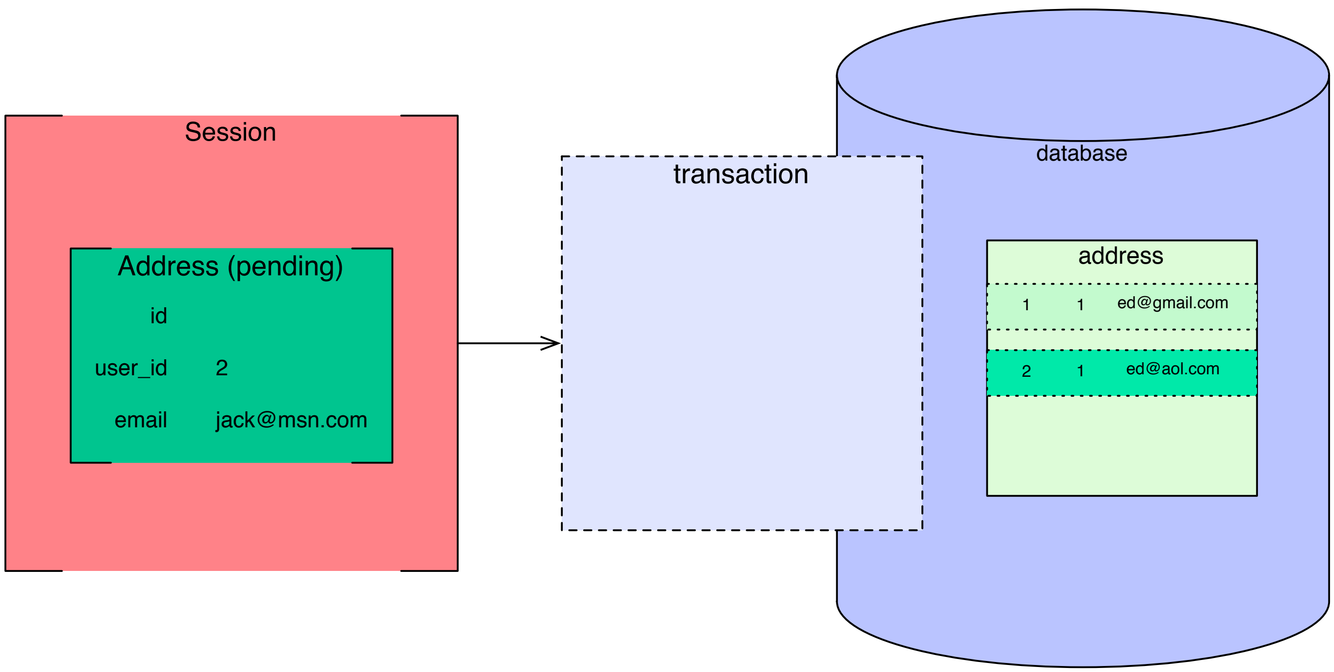
The Object as Row Proxy

An object that's *outside* of the Session, not yet corresponding to any row, is said to be **transient.**



The Object as Row Proxy

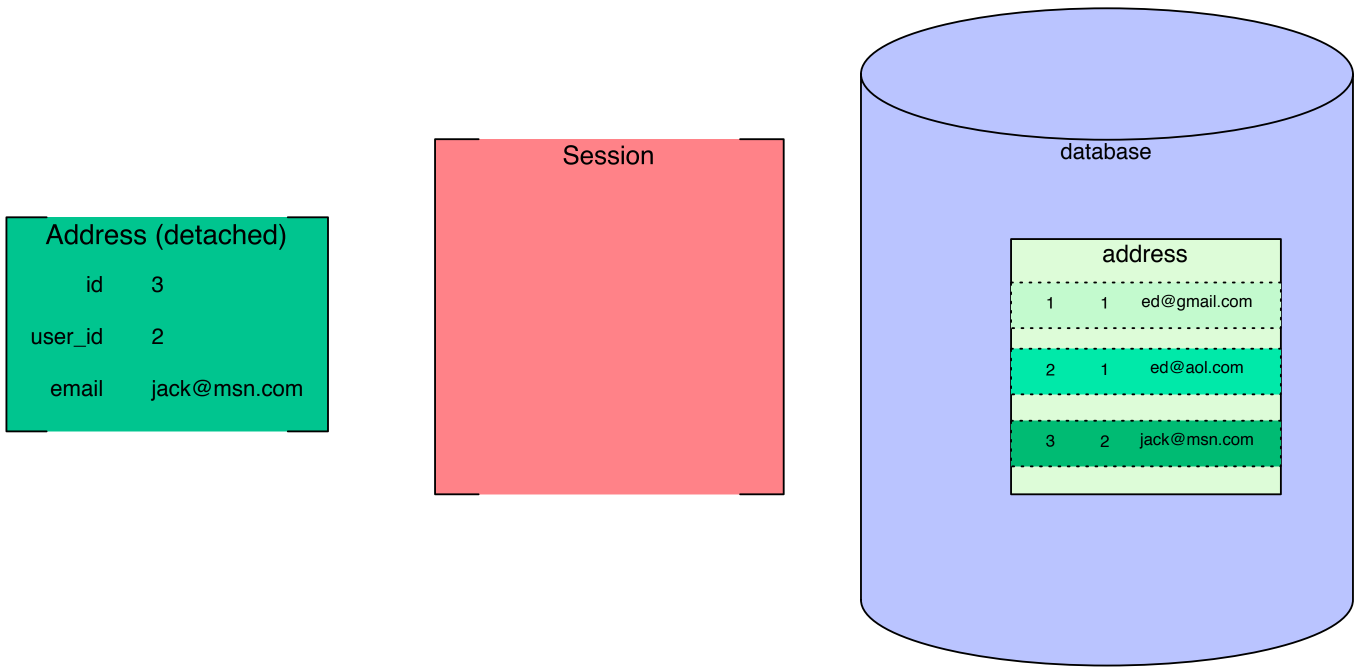
An object that's *inside* of the Session, but not yet corresponding to any row, is said to be **pending.**



The Object as Row Proxy

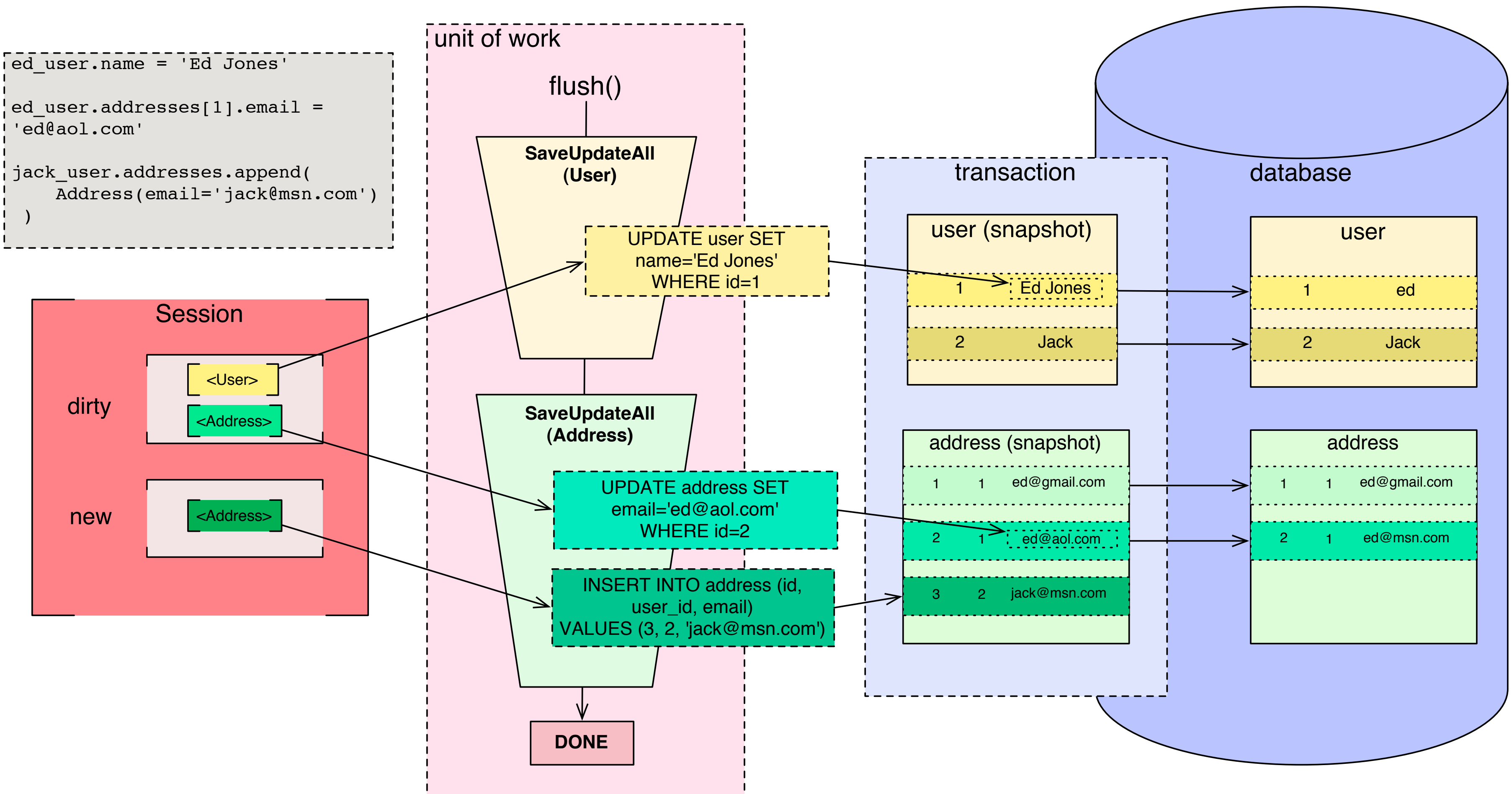
A previously persistent object that's no longer associated with a Session is said to be **detached.**

Detachment is useful for caching, but not much else.



Unit of Work

Unit of work **lazily flushes** only those rows/columns that have changed, ordering to maintain consistency.



Where'd the Session Come from?

- Unit of work, identity map discussed in Martin Fowler, *Patterns of Enterprise Architecture*
- Hibernate for Java largely responsible for developing Session concepts
- Java Persistence Architecture (JSR-220) specifies a similar model, largely driven by Hibernate
- SQLAlchemy moved to a stricter, more correct model in 0.5 through observation of the Storm ORM for Python

Watching the Session
Solve those Issues

Session

Objects are stored in an **identity map**

```
def user_process_one(session):
    user = session.query(User).get(5)
    user.name = 'Jack Jones'
    return user

def user_process_two(session):
    user = session.query(User).get(5)
    if user.name == 'Jack Jones':
        address = Address(email='jack@gmail.com', user=user)
        session.add(address)
    return user

# both functions get the same user
user1 = user_process_one(session)
user2 = user_process_two(session)
session.commit()
```

Session

The **unit of work** pattern aggregates changes and emits as needed

```
session = Session()
for user_record in datafile:
    user = User(name=user_record.username)
    session.add(user)  # no INSERT here

    for entry in user_record.entries:
        if entry.type == 'A':
            address = Address(user=user)
            address.email = entry.email
            session.add(address)  # no INSERT here

        elif entry.type == 'U':
            # changes aggregated in memory.
            user.field1 = entry.field1
            user.field2 = entry.field2

    session.flush()  # optional, will flush this user

session.commit()  # flushes everything still pending
```

Session

Data is expired when transactions, always explicit, are ended - hence no stale data

```
session1 = Session()
user1 = session1.query(User).filter_by(id=5).one()
user1.name = 'New Name'
session1.commit()

session2 = Session()
user2 = session2.query(User).filter_by(id=5).one()
user2.name = 'Some Other Name'
session2.commit()

# user1 was expired by the commit, reloads here
assert user1.name == 'Some Other Name'

# change user2 ...
user2.name = 'Yet Another Name'
session2.rollback()

# user2 was expired by the rollback, reloads here
assert user2.name == 'Some Other Name'
```

Session

Objects proxying to other transactions aren't accepted

```
queue = Queue.Queue()

def user_producer():
    session = Session()
    for record in data:
        user = session.query(User).\
                 filter_by(name=record.username).first()
        if user is None:
             session.add(User(name=record.username))
        queue.put(user)
    session.commit()

def user_consumer():
    while True:
        user = queue.get()
        session = Session()
        if user.status == 'D':
             session.delete(user)  # raises an exception, this user
                                   # proxies a row from a different
                                   # transaction. Code fails
                                   # unconditionally.
        session.commit()
        queue.task_done()
```

"Live" Session Demo

User/Address Model

```python
class User(Base):
    __tablename__ = "user"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    addresses = relationship("Address")

class Address(Base):
    __tablename__ = "address"

    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
```

Example Code

```
u1 = User(name="ed")

u1.addresses = [
    Address(email="ed@ed.com"),
    Address(email="ed@gmail.com"),
    Address(email="edward@python.net"),
]

session = Session()

session.add(u1)
session.commit()

u1.addresses[1].email = "edward@gmail.com"
session.commit()
```

SQLAlchemy

We're done ! Hope this was enlightening.

http://www.sqlalchemy.org