Hand Coded Applications with SQLAlchemy

What's a Database?

- We can put data in, get it back out.
- Data is stored as records/rows/documents/etc.
- Records/rows are composed of sets of attributes.
- Queries allow us to find records that have specific attributes.

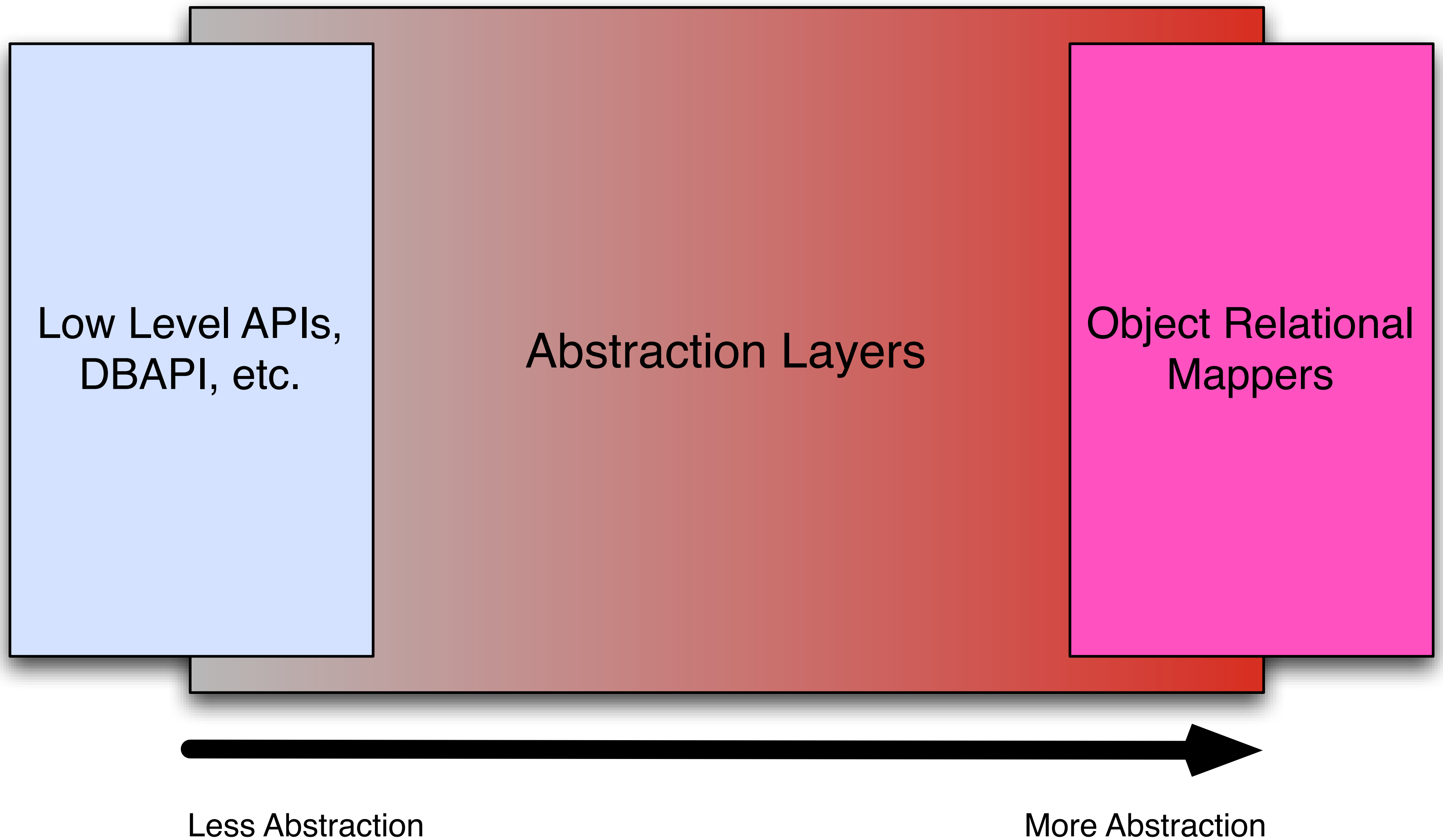
What's a Relational Database?

- Fundamental storage unit is the column, composed into rows, composed into tables.
- Rows across multiple tables can be transformed into new rows at query time using joins.
- Rows can be formed into "derived tables" using subqueries.
- Set operations, aggregates, grouping, recursive queries, window functions, triggers, functions/SPs, etc.
- Transactional guarantees (i.e. the ACID model)

How do we talk to relational databases?

- Database APIs, i.e. DBAPI in Python
- Abstraction layers
- Object relational mappers

How do we talk to databases?



What's an ORM?

- Automates persistence of domain models into relational schemas
- Provide querying and SQL generation in terms of a domain model
- Translates relational result sets back into domain model state
- Mediates between object-oriented and relational geometries (relationships, inheritance)

How much "abstraction" should an ORM provide?

- Conceal details of how data is stored and queried?
- Conceal that the database itself is relational?
- Should it talk to nonrelational sources (MongoDB, DBM) just like a SQL database?
- These questions ask to what degree we should be "hiding" things.

Problems with ORM Abstraction Considered as "Hiding"

- SQL language is relational - joins, set operations, derived tables, aggregates, grouping, etc.
- Ability to organize and query for data in a relational way is the primary feature of relational databases.
- Hiding it means you no longer have first-class access to those features.
- Relational database is under-used, mis-used
- "Object Relational Impedance Mismatch"

We don't want "hiding". We want "automation".

- We are best off when we design and control the schema/query side as well as how our object model interacts with it.
- We still need tools to automate this process.
- Explicit decisions + automation via tools = "Hand Coded".

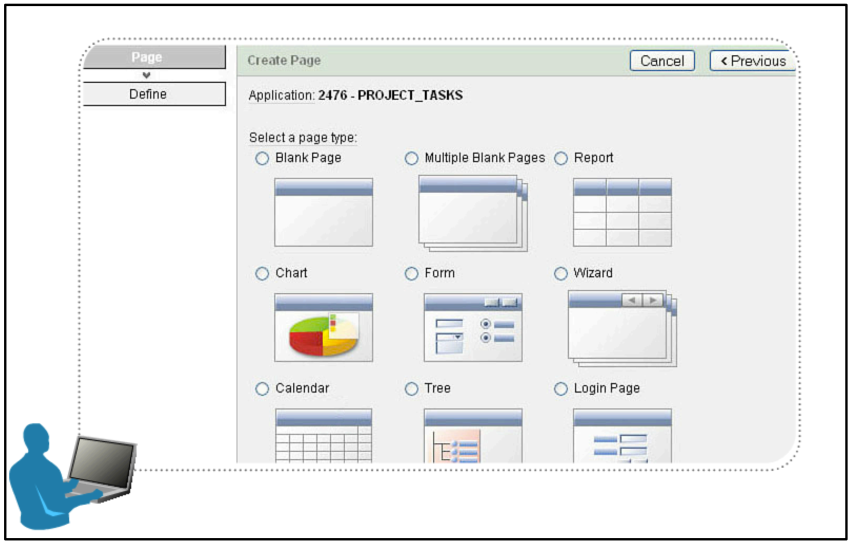
SQLAlchemy and the Hand Coded Approach

Hand Coded ?

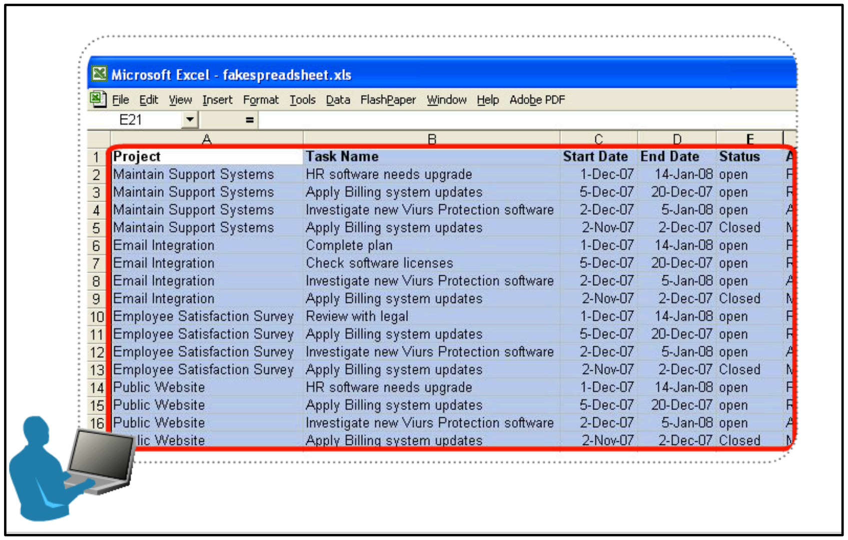

mike bayer @zzzeek
7 Oct
possible pycon talk: "Hand-coded applications with SQLAlchemy". Or "hand-crafted" ? I feel like "handcrafted" is hackneyed these days
Reply
Doug Hellmann
@doughellmann
Following
@zzzeek How would they be made other than by hand?
5:25 PM - 7 Oct 11 via Twitter for Mac · Embed this Tweet
Reply
Retweet
Favorite

Not All Apps are Hand Coded!

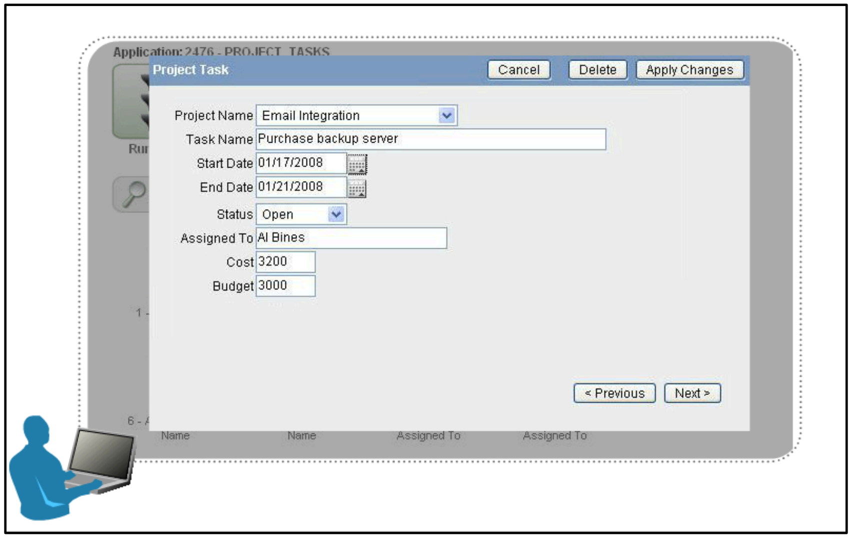
Call the Wizard...



Cut and Paste your data...



Formus Creare!



It's Web Scale!

FAST

For building internet or intranet applications using only a Web browser

SCALABLE

To support growing data and user access

SECURE

Includes built-in access management and data protection

But Zeek, those are corporate GUI wizards, that's not what we do in Python !

- But, when we use tools that:
  - Make schema design decisions for us
  - Conceal the database, relational geometry
  - Give us a "pass" on having to obey ACID
  - It's a step in that direction; we give up control of architecture to a third party.

Hand Coded Means:

- We make decisions.
- We implement and automate those decisions with tools.
- We retain control over the architecture.

Hand Coded is the Opposite of:

- Apps written by "wizards" - **Obvious**
- Apps that rely very heavily on "plugins", third party apps - **Less Obvious**
- Using APIs that make implementation decisions - **Subtle**

Hand Coded does **not** mean:

- We don't use any libraries.
  - Definitely use these as much as possible!
- We don't use frameworks.
  - Frameworks are great if they don't make things harder!
- We don't use defaults.
  - We make our own "defaults" using self-established conventions.

What are some examples of "implementation decisions" and "hiding"?

- Example 1: the "polymorphic association" pattern
- Example 2: querying approaches that oversimplify relational geometry

Example 1 – Polymorphic Association

Define a simple model representing accounts, assets.

```
class BankAccount(BaseEntity):
    owner = String
    identifier = String

class PortfolioAsset(BaseEntity):
    owner = String
    symbol = String
```

Example 1 – Polymorphic Association

Table definitions for these are:

portfolio_asset	
id	INTEGER (PK)
owner	VARCHAR
symbol	VARCHAR

bank_account	
id	INTEGER (PK)
owner	VARCHAR
identifier	VARCHAR

Example 1 – Polymorphic Association

Now add a collection of "financial transactions" to each:

```python
from magic_library import GenericReference

class FinancialTransaction(BaseEntity):
    amount = Numeric
    timestamp = DateTime

class BankAccount(BaseEntity):
    owner = String
    identifier = String
    transactions = GenericReference(FinancialTransaction)

class PortfolioAsset(BaseEntity):
    owner = String
    symbol = String
    transactions = GenericReference(FinancialTransaction)
```

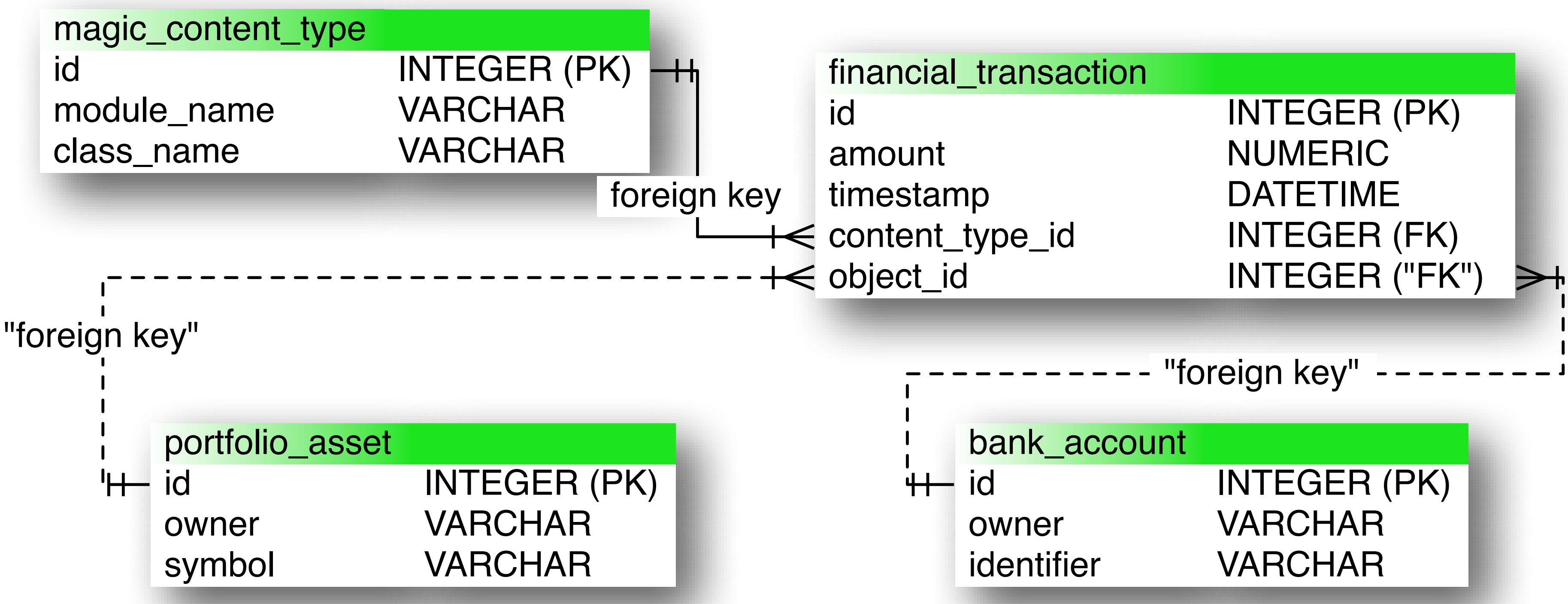
Example 1 – Polymorphic Association

Usage might look like:

```
some_bank_account.transactions = [
        FinancialTransaction(100, datetime(2011, 10, 15, 12, 57, 7)),
        FinancialTransaction(-50, datetime(2011, 11, 2, 8, 0, 0))
]

some_portfolio_asset.transactions = [
        FinancialTransaction(525, datetime(2011, 9, 18, 17, 52 5)),
        FinancialTransaction(54.12, datetime(2011, 9, 29, 15, 8, 7)),
        FinancialTransaction(-10.04, datetime(2011, 10, 2, 5, 30, 17))
]
```

Example 1 – Polymorphic Association

What did `GenericReference` just build for us?



Example 1 – Polymorphic Association

What did `GenericReference` just build for us?

```
INSERT INTO magic_content_type (id, module_name, class_name) VALUES (
    (1, "someapp.account", "BankAccount"),
    (2, "someapp.asset", "PortfolioAsset"),
)

INSERT INTO financial_transaction (id, amount, timestamp, object_id,
content_type_id) VALUES (
    (1, 100, '2011-10-15 12:57:07', 1, 1),
    (2, -50, '2011-11-02 08:00:00', 1, 1),
    (3, 525, '2011-09-18 17:52:05', 2, 2),
    (4, 54.12, '2011-09-29 15:08:07', 2, 2),
    (5, -10.04, '2011-10-02 05:30:17', 2, 2)
)
```

Implicit Design Decisions

- Added "magic_" tables to our schema.
- Python source code (module and class names) stored as data, hardwired to app structure.
- Storage of transaction records are in one monolithic table, as opposed to table-per-class, other schemes.
- Schema design is not constrainable. The application layer, not the database, is responsible for maintaining consistency.

Polymorphic Association – SQLAlchemy's Approach

- SQLAlchemy's approach encourages us to specify how tables are designed and mapped to classes explicitly.
- We use regular Python programming techniques to establish composable patterns.
- This approach expresses our exact design fully and eliminates boilerplate at the same time.

Composable Patterns

Start with a typical SQLAlchemy model:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class BankAccount(Base):
    __tablename__ = 'bank_account'

    id = Column(Integer, primary_key=True)
    identifier = Column(String(38), nullable=False, unique=True)
    owner = Column(String(30), nullable=False)

class PortfolioAsset(Base):
    __tablename__ = 'portfolio_asset'

    id = Column(Integer, primary_key=True)
    symbol = Column(String(30), nullable=False, unique=True)
    owner = Column(String(30), nullable=False)
```

Composable Patterns

Use Base classes to establish conventions common to all/most mapped classes.

```python
import re
from sqlalchemy.ext.declarative import declared_attr

class Base(object):

    @declared_attr
    def __tablename__(cls):
        # convert from CamelCase to words_with_underscores
        name = cls.__name__
        return (
            name[0].lower() +
            re.sub(r'([A-Z])',
            lambda m:"_" + m.group(0).lower(), name[1:])
        )

    # provide an "id" column to all tables
    id = Column(Integer, primary_key=True)

Base = declarative_base(cls=Base)
```

Composable Patterns

Use mixins and functions to define common patterns

```
class HasOwner(object):
    owner = Column(String(30), nullable=False)

def unique_id(length):
    return Column(String(length), nullable=False,
                                          unique=True)
```

Composable Patterns

Now the same model becomes succinct

```
class BankAccount(HasOwner, Base):
    identifier = unique_id(38)

class PortfolioAsset(HasOwner, Base):
    symbol = unique_id(30)
```

HasTransactions Convention

Define a convention for the polymorphic association, using table-per-class.

```
class TransactionBase(object):
    amount = Column(Numeric(9, 2))
    timestamp = Column(DateTime)

    def __init__(self, amount, timestamp):
        self.amount = amount
        self.timestamp = timestamp
```

HasTransactions Convention

Define a convention for the polymorphic association, using table-per-class.

```
class HasTransactions(object):

    @declared_attr
    def transactions(cls):
        cls.Transaction = type(
            # create a new class, i.e. BankAccountTransaction
            "%sTransaction" % cls.__name__,
            (TransactionBase, Base,),
            dict(
                # table name: "bank_account_transaction"
                __tablename__ = '%s_transaction' %
                                     cls.__tablename__,

                # "bank_account_id REFERENCES (bank_account.id)"
                parent_id = Column('%s_id' % cls.__tablename__,
                        ForeignKey("%s.id" % cls.__tablename__),
                        nullable=False)
            )
        )
        # relate HasTransactions -> Transaction
        return relationship(cls.Transaction)
```

HasTransactions Convention

Apply HasTransactions to the Model

```python
class BankAccount(HasTransactions, HasOwner, Base):
    identifier = unique_id(38)

class PortfolioAsset(HasTransactions, HasOwner, Base):
    symbol = unique_id(30)
```

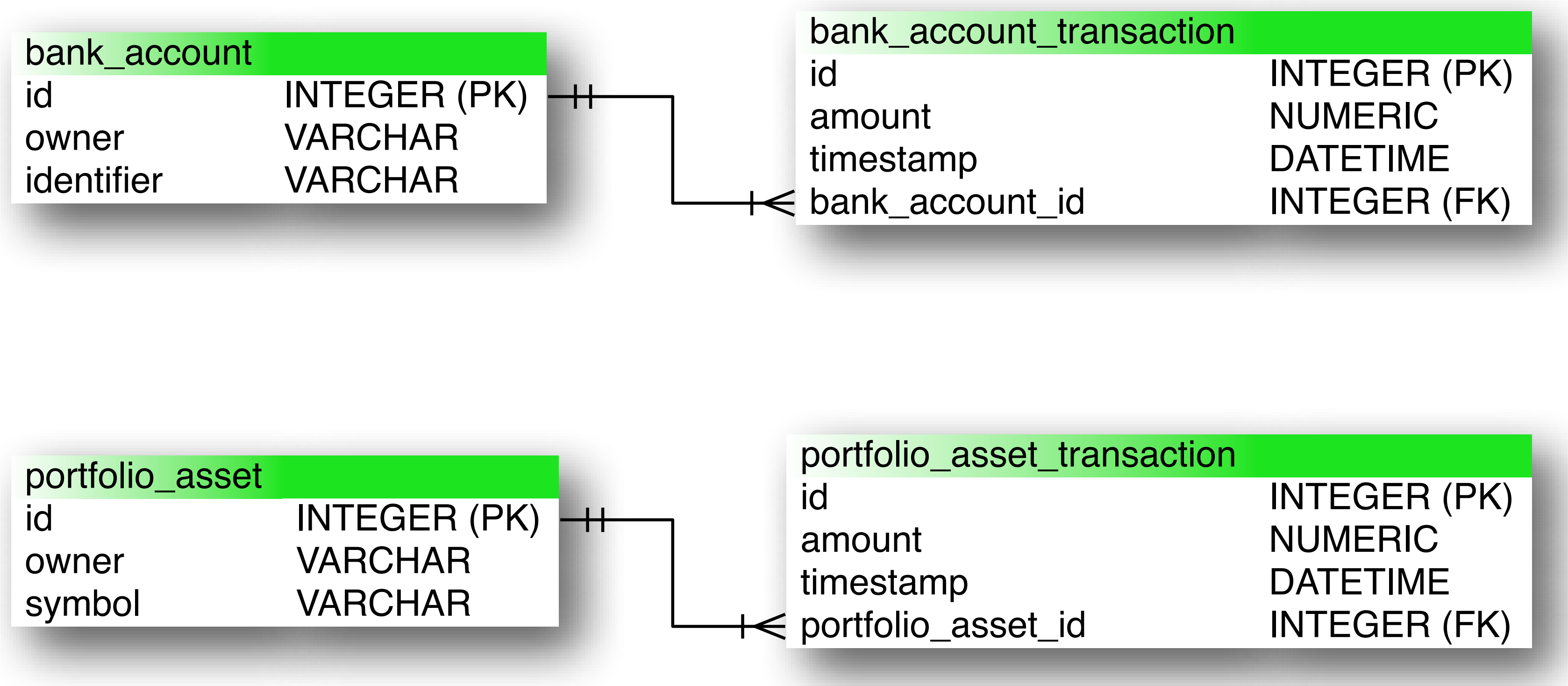
`HasTransactions` Convention

Rudimental usage is similar.

```
some_bank_account = BankAccount(identifier="1234", owner="zzzeek")

some_bank_account.transactions = [
  BankAccount.Transaction(100, datetime(2011, 10, 15, 12, 57, 7)),
  BankAccount.Transaction(-50, datetime(2011, 11, 2, 8, 0, 0))
]

some_portfolio_asset = PortfolioAsset(
                   identifier="AAPL", owner="zzzeek")

some_portfolio_asset.transactions = [
  PortfolioAsset.Transaction(525, datetime(2011, 9, 18, 17, 52 5)),
  PortfolioAsset.Transaction(54.12, datetime(2011, 9, 29, 15, 8, 7)),
  PortfolioAsset.Transaction(-10.04, datetime(2011, 10, 2, 5, 30, 17))
]
```

HasTransactions Convention

What's the schema?



Polymorphic Association – Summary

- Composed `HasTransactions` using a well-understood recipe.
- Used our preferred naming/structural conventions.
- Uses constraints and traditional normalized design properly.
- Data in separate tables–per–parent (other schemes possible too).
- Data not hardcoded to application structure or source code

Why not improve `GenericReference` to support these practices?

- `GenericReference` would need to present various modes of behavior in the form of more options and flags, leading to a complex configuration story.
- Once we know the polymorphic association recipe, it becomes trivial and self documenting. It's too simple to warrant introducing configurational complexity from outside.

Example 2 – Exposing Relational Geometry

Query for the balance of an account, as of a certain date.

```
# A "hide the SQL" system might query like this:

some_bank_account.transactions.sum("amount").
        filter(lessthan("timestamp", somedate))

# Obvious SQL:

SELECT SUM(amount) FROM bank_account_transactions
WHERE
    bank_account_id=2 AND
    timestamp <= '2010-09-26 12:00:00'
```

Example 2 - Exposing Relational Geometry

- The "hide the SQL" system easily applied an aggregate to a single field on a related collection with a simple filter.
- But now I want:
  - A report of average balance per month across all accounts.

Example 2 - Exposing Relational Geometry

- Because our model stores data as individual transaction amounts, we need to use subqueries and/or window functions to produce balances as a sum of amounts.
- In SQL, we build queries like these incrementally, referencing relational structures explicitly and building from the inside out.
- If our tools prevent us from doing this, we either need to bypass them, or load the rows into memory (which doesn't scale).

Example 2 - Exposing Relational Geometry

- SQLAlchemy's query model explicitly exposes the geometry of the underlying relational structures.
- Like "power steering" for SQL. Doesn't teach you how to drive!
- Developer should be aware of the SQL being emitted. SQLAlchemy makes this easy via logging or "echo" flag.
- Just like your car has windows to see where you're going!

Build a Query from the Inside Out

Start with a query that extracts all the start/end dates of each month in the `bank_account_transaction` table:

```
SELECT MIN(timestamp) AS min,
       MAX(timestamp) AS max,
       EXTRACT (year FROM timestamp) AS year,
       EXTRACT (month FROM timestamp) AS month
FROM bank_account_transaction
GROUP BY year, month
ORDER BY year, month
```

Build a Query from the Inside Out

Sample month ranges:

min	max	year	month
2009-03-08 10:31:16	2009-03-28 11:03:46	2009	3
2009-04-05 08:02:30	2009-04-30 01:06:23	2009	4
2009-05-02 22:38:42	2009-05-31 16:03:38	2009	5
2009-06-08 23:17:23	2009-06-30 03:24:03	2009	6
2009-07-04 09:47:18	2009-07-31 21:20:08	2009	7
2009-08-04 22:07:11	2009-08-30 12:20:17	2009	8
2009-09-01 05:44:06	2009-09-30 05:18:24	2009	9
2009-10-01 13:30:27	2009-10-29 12:47:23	2009	10
2009-11-02 08:30:03	2009-11-29 13:54:39	2009	11
2009-12-01 14:25:58	2009-12-28 20:01:35	2009	12
2010-01-01 11:55:21	2010-01-30 19:49:28	2010	1
2010-02-01 12:25:38	2010-02-26 14:18:07	2010	2

...

Build a Query from the Inside Out

The other half of the query will use a "window" function – evaluates an aggregate function as rows are processed.

```
SELECT
    bank_account_id,
    timestamp,
    amount,
    SUM(amount) OVER (
        PARTITION BY bank_account_id
        ORDER BY timestamp
    )
FROM bank_account_transaction
```

Build a Query from the Inside Out

Sample data from the "window":

```
bank_account_id  |      timestamp       |  amount   |   sum
-----------------+----------------------+-----------+----------
               1 | 2009-05-19 23:28:22 |  7925.00 |  7925.00
               1 | 2009-06-17 13:24:52 |   146.00 |  8071.00
               1 | 2009-06-18 11:49:32 |  2644.00 | 10715.00
               ...
               2 | 2009-04-09 14:36:48 |  5894.00 |  5894.00
               2 | 2009-04-10 13:20:50 |  1196.00 |  7090.00
               2 | 2009-05-06 21:07:26 | -3485.00 |  3605.00
               ...
               3 | 2009-03-18 21:21:11 |  6648.00 |  6648.00
               3 | 2009-04-17 15:43:31 |   711.00 |  7359.00
               3 | 2009-04-23 06:41:20 | -1775.00 |  5584.00
               ...
```

Build a Query from the Inside Out

Join these two queries together:

```
SELECT year, month, avg(balances.balance) FROM
    (SELECT MIN(timestamp) AS min,
            MAX(timestamp) AS max,
            EXTRACT (year FROM timestamp) AS year,
            EXTRACT (month FROM timestamp) AS month
    FROM bank_account_transaction
    GROUP BY year, month) AS month_ranges
JOIN (SELECT timestamp,
        SUM(amount) OVER (
            PARTITION BY bank_account_id
            ORDER BY timestamp
        ) AS balance
    FROM bank_account_transaction
    ) AS balances
ON balances.timestamp
  BETWEEN month_ranges.min AND month_ranges.max
GROUP BY year, month ORDER BY year, month
```

Build a Query from the Inside Out

Final Result

```
 year | month |   avg
------+-------+----------
 2009 |     3 |  5180.75
 2009 |     4 |  5567.30
 2009 |     5 |  9138.33
 2009 |     6 |  8216.22
 2009 |     7 |  9889.50
 2009 |     8 | 10060.92
 2009 |     9 | 10139.81
 2009 |    10 | 15868.20
 2009 |    11 | 16562.52
 2009 |    12 | 17302.37
 ...
```

Build a Query from the Inside Out

- Now we'll build this in SQLAlchemy.
- SQLAlchemy provides the same "inside out" paradigm as SQL itself.
- You think in terms of SQL relations and joins in the same way as when constructing plain SQL.
- SQLAlchemy can then apply automated enhancements such as eager loading, row limiting, further relational transformations.

Build a `Query()` from the Inside Out

All the start/end dates of each month in the `bank_account_transaction` table:

```
from sqlalchemy import func, extract

Transaction = BankAccount.Transaction

month_ranges = session.query(
    func.min(Transaction.timestamp).label("min"),
    func.max(Transaction.timestamp).label("max"),
    extract("year", Transaction.timestamp).label("year"),
    extract("month", Transaction.timestamp).label("month")
  ).group_by(
      "year","month"
  ).subquery()
```

Build a `Query()` from the Inside Out

All balances on all days via window function:

```
all_balances_and_timestamps = session.query(
                    Transaction.timestamp,
                    func.sum(Transaction.amount).over(
                       partition_by=Transaction.parent_id,
                       order_by=Transaction.timestamp
                    ).label("balance")
                ).subquery()
```

Build a `Query()` from the Inside Out

Join the two together:

```
avg_balance_per_month = \
    session.query(
        month_ranges.c.year,
        month_ranges.c.month,
        func.avg(all_balances_and_timestamps.c.balance)).\
      select_from(month_ranges).\
      join(all_balances_and_timestamps,
        all_balances_and_timestamps.c.timestamp.between(
            month_ranges.c.min, month_ranges.c.max)
    ).group_by(
      "year", "month"
    ).order_by(
      "year", "month"
    )
```

Build a `Query()` from the Inside Out

The Result

```
for year, month, avg in avg_balance_per_month:
    print year, month, round(avg, 2)

2009      3     5180.75
2009      4     5567.3
2009      5     9138.33
2009      6     8216.22
2009      7     9889.5
2009      8     10060.93
2009      9     10139.82
2009      10    15868.2
2009      11    16562.53
2009      12    17302.38
...
```

Build a `Query()` from the Inside Out

The SQL

```
SELECT
    anon_1.year AS anon_1_year,
    anon_1.month AS anon_1_month,
    avg(anon_2.balance) AS avg_1 FROM (
    SELECT
      min(bank_account_transaction.timestamp) AS min,
      max(bank_account_transaction.timestamp) AS max,
      EXTRACT(year FROM bank_account_transaction.timestamp:: timestamp) AS year,
      EXTRACT(month FROM bank_account_transaction.timestamp::timestamp) AS month
    FROM bank_account_transaction
    GROUP BY year, month
) AS anon_1 JOIN (
    SELECT
        bank_account_transaction.bank_account_id AS bank_account_id,
        bank_account_transaction.timestamp AS timestamp,
        sum(bank_account_transaction.amount) OVER (
            PARTITION BY bank_account_transaction.bank_account_id
            ORDER BY bank_account_transaction.timestamp
        ) AS balance
    FROM bank_account_transaction
) AS anon_2 ON anon_2.timestamp BETWEEN anon_1.min AND anon_1.max
GROUP BY year, month ORDER BY year, month
```

Hand Coded - Summary

- The developer retains control over the relational form of the target data.
- Schema design decisions are all made by the developer. Tools shouldn't make decisions.
- SQLA provides a rich, detailed vocabulary to express and automate these decisions.
- Developer creates patterns and conventions based on this vocabulary.
- Relational geometry remains an explicit concept complementing the object model.

"Leaky Abstraction"

- This term refers to when an abstraction layer exposes some detail about what's underneath.
- Does SQLAlchemy's schema design paradigm and `Query()` object exhibit this behavior?
- You bet!
- *All non-trivial abstractions, to some degree, are leaky.* - Joel On Software
- SQLAlchemy accepts this reality up front to create the best balance possible.

Hand Coded...

...vs. Design by 3rd Party

Hand Coded produces accurately targeted, long lasting designs that resist technical debt

SQLAlchemy

We're done !
Hope this was
enlightening.

http://www.sqlalchemy.org