BloxAlchemy

A toolkit for using logical databases with Python

PREDICTIX

Outline

- Datalog and SQL
- BloxAlchemy Architecture
- Metadata
- SQL Layer & Logic Generation
- ORM Layer
- BloxSoup

What is Datalog?

- Declarative, logical language
  - Uses predicate logic
  - Order of statements does not matter
- Predicates and Facts
  - is_parent is a predicate (think of it as a class)
  - is_parent(Rick, Matthew) is a fact (think of it as an instance)
- Database Management System
  - Predicates and facts are persistent
  - Updates to fact database are done in transactions

Datalog Samples

- Declare an *entity* with a *refmode*
  - user(u), user:user_name(u:n) -> string(n).
  - department(d), department:name(d:n) -> string(n).
- Declare a *predicate*
  - user:department(u;d) -> user(u),department(d).
  - sales(pr,st,wk;s) -> product(pr), store(st), week(wk), float[32](s).
- Declare a predicate with a *rule*
  - department:num_users(d;n) -> department(d),int[32](n).
  - department:num_users(d;n) <- agg<<n=count()>>(
    department(d),user:department(u;d)).

Datalog vs. SQL (30,000 foot view)

SQL

- Tables *may* have keys
- Tables may have multiple value columns
- Query results are generally unkeyed
- Foreign keys, views, and triggers enforce integrity

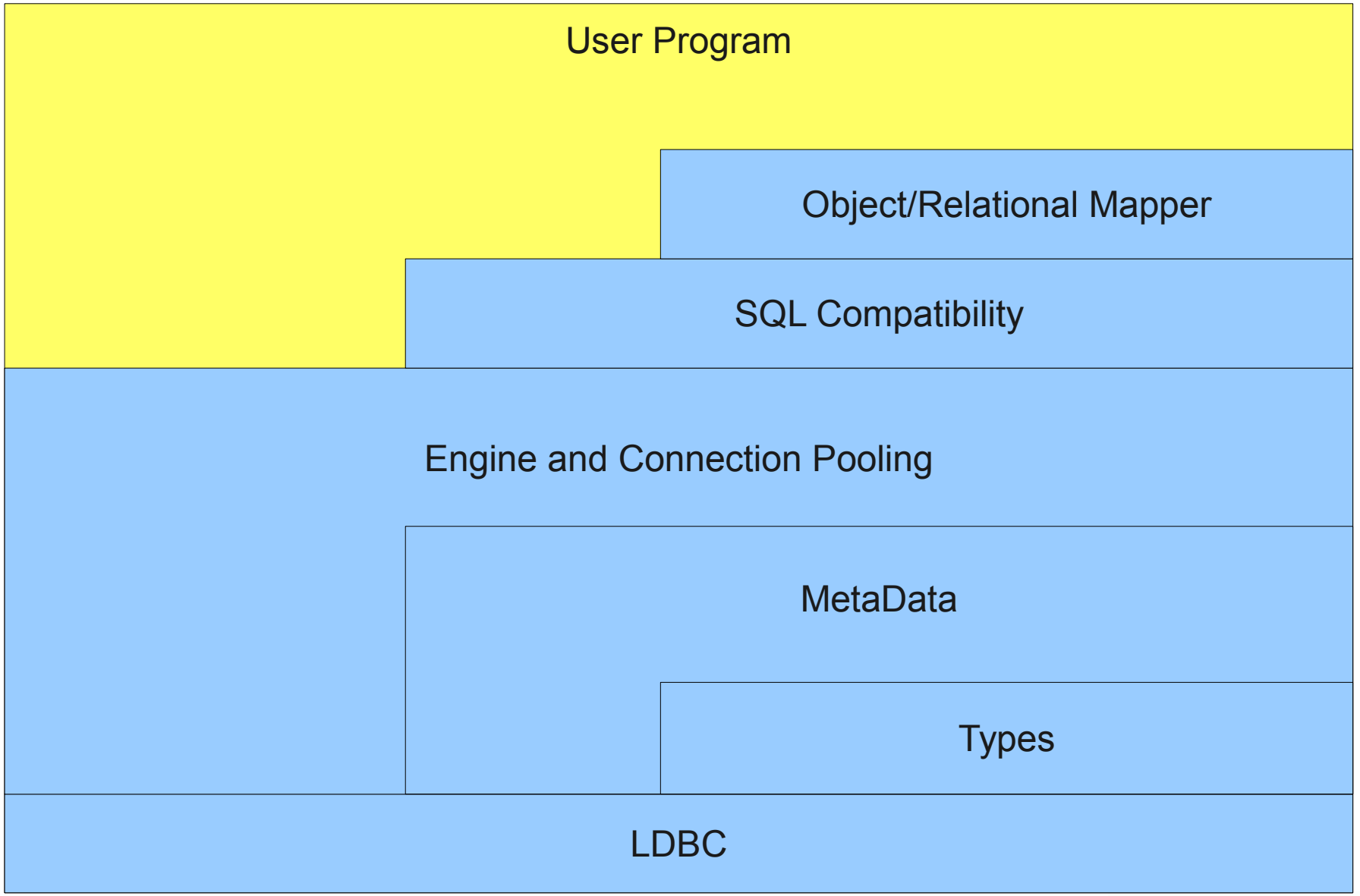
Datalog

- Predicates *must* have keys
- Predicates may have *zero or one* value columns
- Query results are predicates and must have keys
- Keys and derivation rules enforce integrity

Datalog and Python

- LogicBlox (who makes our Datalog software) provides:
  - LDBC – minimal interface to use Datalog modeled after ODBC and JDBC
  - LogicBlox.py – SWIG-generated LDBC wrapper
- LDBC interface
  - Datalog expressed as strings (unsafe!)
  - Facts retrieved from predicates via iterators
    - Support for iterating in sorted order

BloxAlchemy Architecture



BloxAlchemy Metadata

- Purpose: Map SQL concepts onto Datalog implementation

- SQL tables translate to...
  - Entity – represents an entity and all predicates that are keyed by that entity alone
  - PredicateGroup – represents a collection of predicates keyed by the same set of entities

SQL Layer and Logic Generation

- Convert this:

```
select([user.c.user_name, user.c.password],
    whereclause=and_(
        user.c.department==department.c.self,
        department.c.name=='development'))
```

- To this:

```
_filter(user,department) <-
  identity:user(user),
  identity:department(department),
  !identity:user:is_null(user),
  !identity:department:is_null(department),
  identity:user:department(user;department),
  identity:department:name(department:name),
  (name = “development”).
_0(user,department;user_name) <-
  _filter(user,department),identity:user:user_name(user:user_name).
_1(user,department;department) <-
  _filter(user,department),identity:user:department(user;department).
```

Logic Generation Approach

- "SQL" Query Structure
  - "Inner" Part
    - Set of inner join tables
    - WHERE clause for inner join condition & filtering
  - "Outer" Part
    - Set of LEFT OUTER JOIN tables with ON clauses
  - List of columns to be returned
    - can come from either inner or outer parts
- Example

```
select([user.c.user_name, department.c.name],
    whereclause=user.c.user_name.like('r%'),
    from_obj=[user],
    outerjoin=[(department, department.c.self==user.c.department)])
```

Logic Generation Approach

- Inner Part: Generate keyspace

```
_filter_inner(user) <- string:like(user_name,"r%"),
    identity:user:user_name(user:user_name),
    identity:user(user),!identity:user:is_null(user).
```

- Outer Part: Add keys for outer tables

```
_exists0(user) <- _filter_inner(user),identity:user:department(user;_).
_outer0(user,department) <- _filter_inner(user),
    !identity:department:is_null(department),
    identity:department(department),
    identity:user:department(user;department).
_outer0(user,department) <- _filter_inner(user),
    !_exists0(user),identity:department:is_null(department).
_filter(user,department) <- _outer0(user, department).
```

Logic Generation Approach

- List of Columns: “Join out” to column data

```
_0(user,department;user_name) <- _filter(user,department),
    identity:user:user_name(user:user_name).
_1(user,department;name) <- _filter(user,department),
    identity:department:name(department:name).
```

- Result set is a Python iterator that yields instances of the BloxRow class
  - Sorting (ORDER BY) handled by LDBC
  - OFFSET and LIMIT handled by BloxAlchemy

ORM Layer

- Map application classes to SQL-layer “tables”
  - Row in a table ==> instance of a class
  - Column in a row ==> instance property
  - 1:N, M:N join ==> instance collection property (“department.users”)
  - N:1 join ==> property that is an instance of another class (“user.department”)
- Manage loading objects from the database and flushing changes to objects back to the database

ORM Example

```
user_group = Predicate('user_group', metadata,
    Column('user', EntityRef('user'), key=True),
    Column('group', EntityRef('group'), key=True))

class Group(object): pass
class Department(object): pass
class User(object): pass

session.mapper(Group, group)
session.mapper(User, user, properties=dict(
    groups=relation(Group, secondary=user_group, backref='users')))
session.mapper(Department, department, properties=dict(
    users=relation(User, backref=backref('department', lazy=False),
                         cascade='all,delete-orphan')))

u0 = User.get('rick')
u0.department
users = User.query.filter(User.c.user_name.like('r%'))
[ u.department for u in users ]
```

ORM Performance Enhancements

- Unit of work pattern
  - Make multiple changes to your objects, all updates to the database will be performed when the session is “flushed”
- Ability to be either “eager” or “lazy” when loading properties
  - Eager – load the property when the instance itself is created
  - Lazy – load the property when it is first accessed
  - By default, columns are eagerly loaded and joins are lazily loaded, but this can be customized on a per-query basis

Schema Discovery with BloxSoup

- Problem
  - Some people don't like to use Python to declare their schema
  - Some predicates are pre-defined by LogicBlox
- Solution
  - Inspect the live database to determine the schema

BloxSoup Example

```
In [1]: from bloxalchemy.extensions.bloxsoup import DataStore

In [2]: soup = DataStore('ldbc:///./test', namespace_path=['identity'])

In [3]: soup.user
Out[3]: <class 'bloxalchemy.extensions.bloxsoup.AutoMappedTable[user]'>

In [4]: soup.user.table
Out[4]: <Entity identity:user>

In [5]: u = soup.user.get('rick')

In [6]: u.department
Out[6]: <AutoMappedTable[identity:department] name='Development'>

In [7]: u.department.users
Out[7]:
[<AutoMappedTable[user] user_name='rick' password='pw2'
user_password_copy='pw2'>,
 <AutoMappedTable[user] user_name='greg' password=None
user_password_copy=None>]
```