SQLAlchemy and PostgreSQL
presented by
Jason Kirtland and Jonathan Ellis

PostgreSQL Conference West 2008
October 12th

SQLAlchemy

The Python SQL Toolkit and Object Relational Mapper

Python

```
if object.method() == 123:
    user = User('new user')
```

SQLAlchemy

- Open Source, MIT License
- Project started in 2005 by Mike Bayer
- Community Project
- Now up to 23 committers, plus patch contributors, plus third-party plugins and products (including a SA driver by IBM!)

Works With

- PostgreSQL, MySQL, SQLite, Firebird, Oracle, MSSQL, Sybase, DB2, Informix, SAPDB, MSAccess
- Runs anywhere Python runs
- CPython 2.4+, Jython soon!

SQLAlchemy Is...

- Very flexible: a toolkit
- *Slightly* opinionated
  - The truth is fungible
  - We like SQL and don't try to hide it
  - SQLAlchemy never generates names
  - Generated SQL always uses bind parameters

In the Toolkit

- Database Connection Management
- Schema Management and Data Types
- SQL Dialects
- SQL Expression Language
- Object Relational Mapper

Connection Management

- Connection pooling
- Unified interface for connect() arguments and bind parameters
- Auto-commit, transactions, two-phase transactions, SAVEPOINTs
- Multi-database

Schema & Data Types

- SQLAlchemy understands the relational model and makes decisions based on its knowledge of your database layout
- Table structure can be specified in code, reflected from a live database, or a combination
- Hand-written SQL can be used instead of defining tables

Schema & Data Types

- Can issue CREATEs, DROPs, if you want
- “Migration” support
- Supports user defined types- Python-side and server-side

A Legacy Join Table

```
CREATE TABLE groupmembers (
   id INTEGER DEFAULT nextval('groupmembers_id_seq'),
   group_id INTEGER NOT NULL DEFAULT 0,
   member_id INTEGER NOT NULL DEFAULT 0,
   PRIMARY KEY (id))

Table('groupmembers', metadata,
      Column('id', Integer, primary_key=True,
             autoincrement=True),
      Column('group_id', Integer, nullable=False,
             default=0),
      Column('member_id', Integer, nullable=False,
             default=0))
```

Wouldn't It Be Nice If...

```
CREATE TABLE groupmembers (
  group_id INTEGER NOT NULL
           REFERENCES groups (id),
member_id INTEGER NOT NULL
           REFERENCES principals (id),
PRIMARY KEY (group_id, member_id))
```

Fungible!

```
CREATE TABLE groupmembers (
   id INTEGER DEFAULT nextval('groupmembers_id_seq'),
   group_id INTEGER NOT NULL DEFAULT 0,
   member_id INTEGER NOT NULL DEFAULT 0,
   PRIMARY KEY (id))
```

```
Table('groupmembers', metadata,
      Column('group_id', Integer,
             ForeignKey('groups.id'),
             nullable=False, primary_key=True)
      Column('member_id', Integer,
             ForeignKey('principals.id'),
             nullable=False, primary_key=True))
```

- Defining tables in python allows you to use them in SQL expressions:

```
tickets.select().where(tickets.c.id == 5)
```

- And it defines the set of columns that will be selected from each row or inserted.

```
row = tickets.select().execute().fetchone()
```

ACK11030

MANUFACTUREPONUM_I	int4
SEQUENCE_I	int4
PPM_I	int4
ITMNUMBER	bpchar (12)
MNFCENTERLC_I	int4
POSITION_NUM	int4
ROUTING	varchar (32)
NORMSNT	numeric (4,3)
XULID	bpchar (1)
XULID_I	int4
RECVDATE	bpchar (10)
RR	bpchar (12)
RR2	bpchar (12)
RR3	bpchar (12)
POSITION_NUM2	int4
QTYBARFYACK	numeric (131089)
UFONUM	int4
XULSEQNUM	int4
PARASEMT_I	int4
PARASEMB_I	int4
PARASEMINTERACK	int4
ASEMINTERACKPAR	bpchar (4)
AKBARANDJEFF	int4

+ 50 more

↓

Legacy Columns

```
ack = Table('ACK110030', metadata,
    Column('ITMNUMBER, Integer, primary_key=True,
           key='id'),
    Column('MNFCENTERLC_I', Integer,
           ForeignKey('MFC43222.id'),
           key='manufacturing_center_id'),
    ...)
```

- Can help make the code much more readable
- You're stuck with the table names, though
- Only need to define the columns you'll use

Legacy Columns

```
zomg = Table('legacy', metadata,

    Column('null', Integer, primary_key=True),

    ...)
```

- Reserved words are quoted automatically in generated SQL

SQL Dialects

- SQLAlchemy drivers?
- Standard SQL
- Vendor SQL
  - RETURNING
  - @>

SQL Expression Language

- Defined tables form the most common basis for expressions and selectables

```
>>> tickets.c.id
Column('id', Integer(), table=<ticket>,
       primary_key=True, nullable=False)

>>> print tickets.select()
SELECT ticket.id, ticket.type, ticket.time,
ticket.changetime, ticket.component, ticket.severity,
ticket.priority, ticket.owner, ticket.reporter,
ticket.cc, ticket.version, ticket.milestone,
ticket.status, ticket.resolution, ticket.summary,
ticket.description, ticket.keywords
FROM ticket
```

```
>>> sel = select([tickets.c.id]).limit(1)
>>> print sel
SELECT ticket.id FROM ticket LIMIT 1

>>> sel.execute().fetchall()
[(1,)]

>>> sel.execute().fetchone().id
1

>>> sel2 = sel.where(tickets.c.resolution != 'fixed')
>>> print sel2
SELECT ticket.id FROM ticket
WHERE ticket.resolution != ? LIMIT 1

>>> sel2.execute().fetchall()
[(16,)]
```

```
>>> tix = (select([tickets.c.summary, components.c.name]).
            select_from(tickets.join(components)).
            where(and_(tickets.c.owner == 'jek',
                       components.c.owner == 'zzzeek')).
            limit(10))


>>> print tix
SELECT ticket.summary, component.name
FROM ticket JOIN component
  ON component.name = ticket.component
WHERE ticket.owner = ? AND component.owner = ?
LIMIT 10


>>> tix.execute().fetchall()
[('allow column default functions access to the ExecutionContext', 'sql'),
 ('col[2], col[1] = col[1], col[2] loses', 'orm'),
 ('complete attributes.unregister_class', 'orm'),
 ('proposal: replace decoration functions with decorate module', 'orm'),
 ('mapper.py raises wrong exception', 'orm'),
 ('cant set association proxy collection to None', 'orm'),
 ('revisit orderinglist for 0.4', 'orm'),
```

Looks Like SQL

```
>>> (select([tickets.c.summary]).
     where(tickets.c.owner=='jek').
     except_(select([tickets.c.summary]).
             where(tickets.c.resolution=='fixed'))).
     execute().fetchone()
('Zope testrunner coverage might be broken by 0.4.2p3',),
```

```
>>> stats = (
      select([tickets.c.resolution,
              func.count().label('count')]).
      group_by(tickets.c.resolution))


>>> print stats
SELECT ticket.resolution, count(*) AS count
FROM ticket GROUP BY ticket.resolution
```

```
>>> stats.execute().fetchall()
[(None, 150L),
 ('duplicate', 41L),
 ('wontfix', 58L),
 ('fixed', 834L),
 ('invalid', 52L),
 ('worksforme', 50L)]
```

- func.anything()

More Column Comparisons

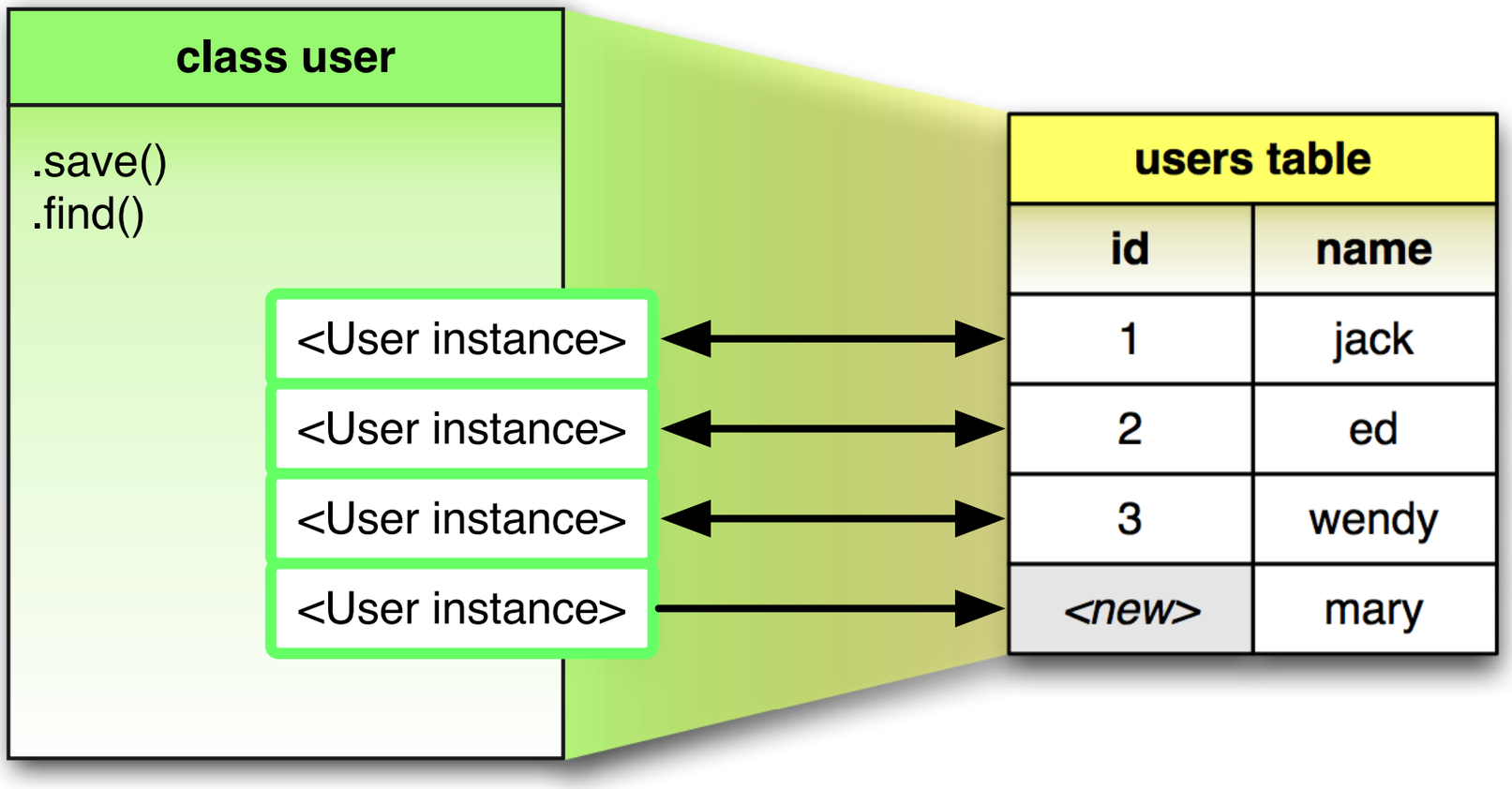
```
>>> print tickets.c.summary == 'abc'
ticket.summary = ?

>>> print tickets.c.changetime.between(back_in_the_day,
                                       func.current_date())
ticket.changetime BETWEEN ? AND CURRENT_DATE

>>> print tickets.c.summary.startswith('foo')
ticket.summary LIKE ? || '%'
```

- WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET, ...
- Assorted JOINs, UNIONs, INTERSECT, EXCEPT, subqueries with correlation control, IN, EXISTS, ...
- INSERT, UPDATE and DELETE too

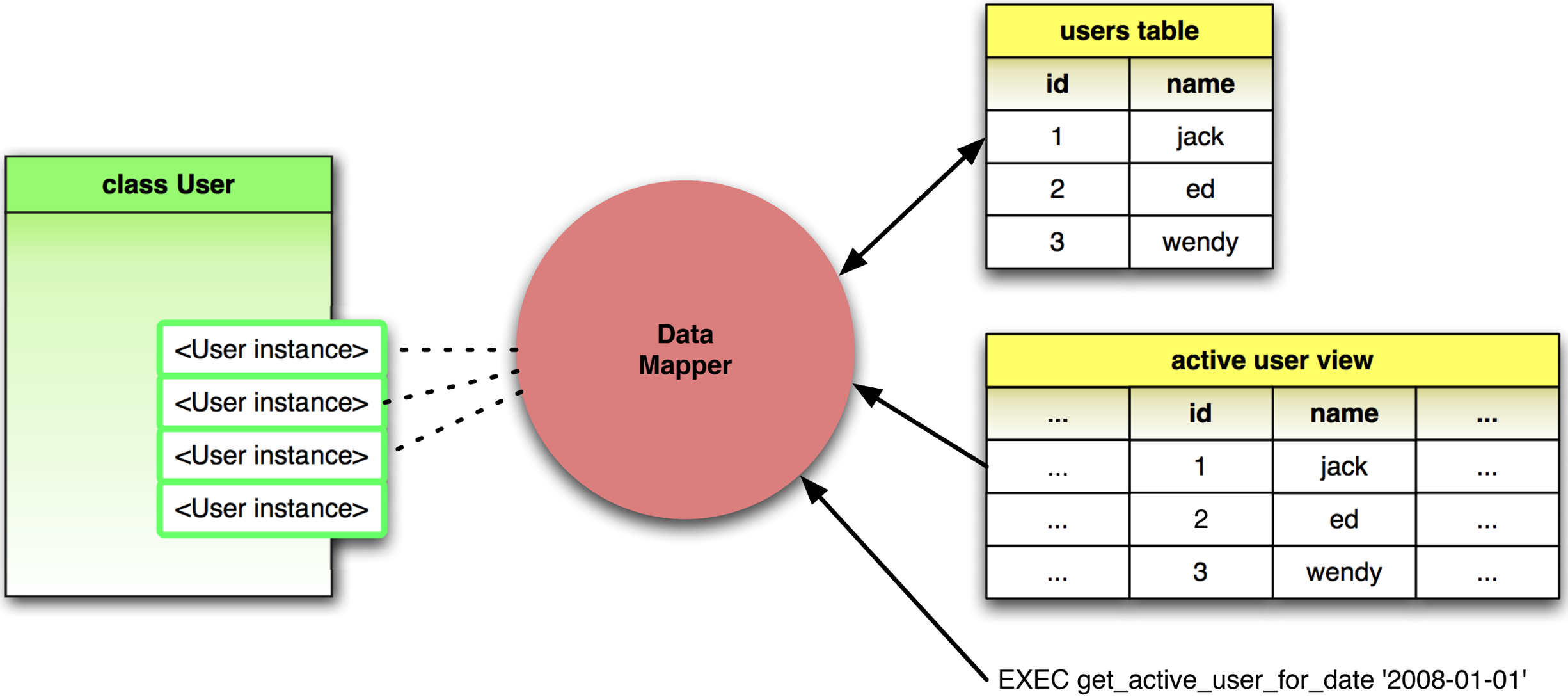
Object Relational Mapper

?

“Active Record”



“Data Mapper”


class User
<User instance>
<User instance>
<User instance>
<User instance>
Data Mapper
users table
id
name
1
jack
2
ed
3
wendy
active user view
...
id
name
...
...
1
jack
...
...
2
ed
...
...
3
wendy
...
EXEC get_active_user_for_date '2008-01-01'

Some ORM Features

- Data Mapper + Any Selectable
- Unit of Work
- Eager / Lazy Loading
- Deferred Properties
- Inheritance Mapping
- Self Referential Mapping
- Sharding

ticket

id	serial
type	text (2147483647)
time	int4
changetime	int4
component	text (2147483647)
severity	text (2147483647)
priority	text (2147483647)
owner	text (2147483647)
reporter	text (2147483647)
cc	text (2147483647)
version	text (2147483647)
milestone	text (2147483647)
status	text (2147483647)
resolution	text (2147483647)
summary	text (2147483647)
description	text (2147483647)
keywords	text (2147483647)

attachment

type	text (2147483647)
id	text (2147483647)
filename	text (2147483647)
size	int4
time	int4
description	text (2147483647)
author	text (2147483647)
ipnr	text (2147483647)

wiki

name	text (2147483647)
version	int4
time	int4
author	text (2147483647)
ipnr	text (2147483647)
text	text (2147483647)
comment	text (2147483647)
readonly	int4

ticket_custom

ticket	int4
name	text (2147483647)
value	text (2147483647)

node_change

rev	text (2147483647)
path	text (2147483647)
node_type	text (2147483647)
change_type	text (2147483647)
base_path	text (2147483647)
base_rev	text (2147483647)

ticket_change

ticket	int4
time	int4
author	text (2147483647)
field	text (2147483647)
oldvalue	text (2147483647)
newvalue	text (2147483647)

milestone

name	text (2147483647)
due	int4
completed	int4
description	text (2147483647)

revision

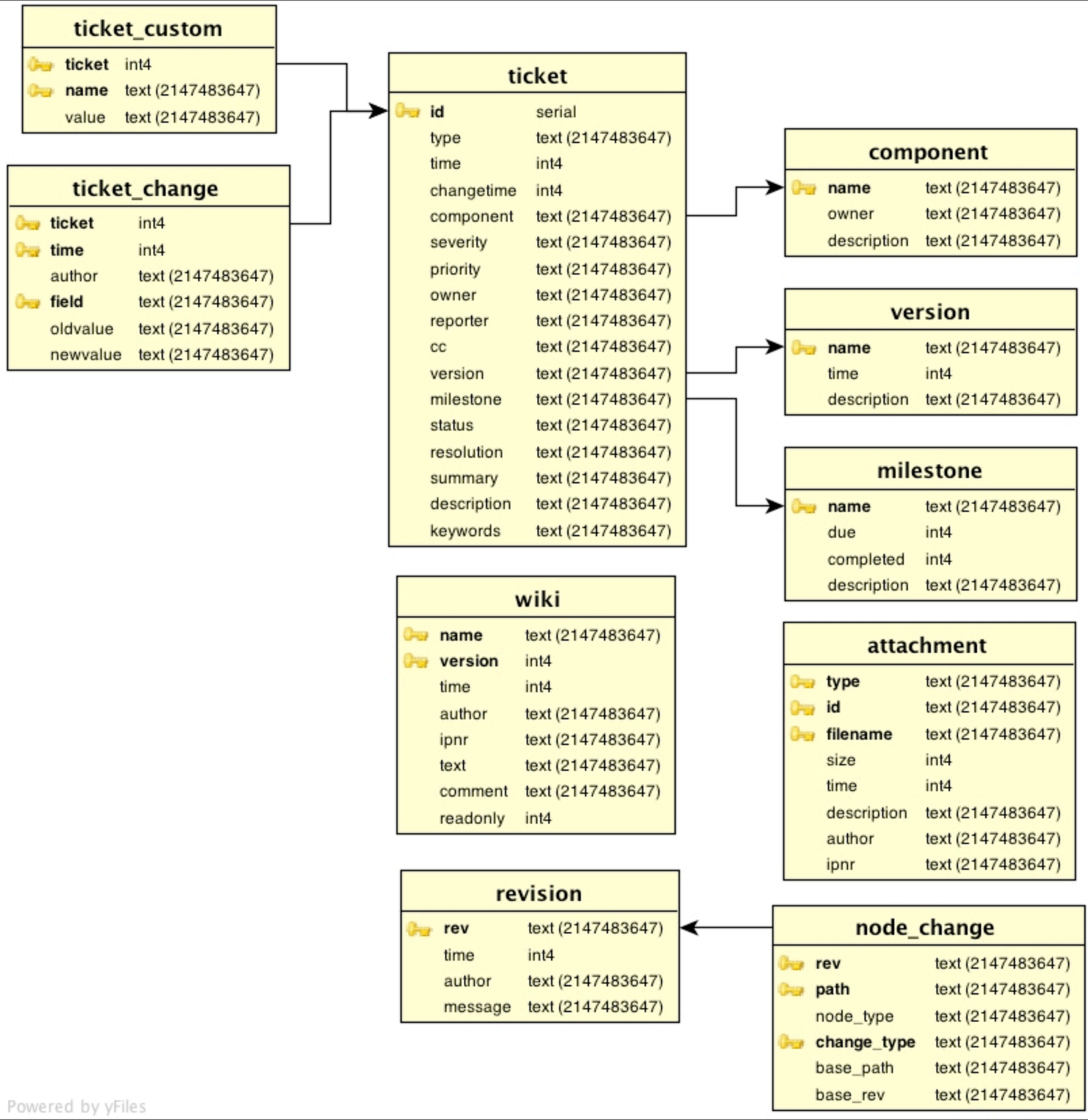
rev	text (2147483647)
time	int4
author	text (2147483647)
message	text (2147483647)

component

name	text (2147483647)
owner	text (2147483647)
description	text (2147483647)

version

name	text (2147483647)
time	int4
description	text (2147483647)



- option 1: purity

```
components = Table('component', metadata,
    Column('name', Text, primary_key=True),
    Column('owner', Text),
    Column('description', Text))

class Component(object):
    pass

mapper(Component, components)
```

- option 2: practicality

```
class Component(Base):
    name = Column(String, primary_key=True)
```

Relations

```
mapper(Ticket, changes, properties={
    'component': relation(Component),
    'version': relation(Version),
    'milestone': relation(Milestone),
    'changes': relation(Change, backref='ticket'),
    'custom_fields': relation(CustomField)
    })
```

session

- A context for interacting with the database
  - fetching objects
  - tracking changes and additions
  - writing changes back
- One per batch of work

session

- Flexible transaction handling
  - Can auto-commit or explicit commit
  - Two-phase transactions
  - Can synchronize object state to transactional state

ORM queries

Query objects

```
>>> session.query(Ticket)
<sqlalchemy.orm.query.Query object at ...>

>>> print _
SELECT ticket.id AS ticket_id,
       ticket.type AS ticket_type,
       ...,
       ticket.keywords AS ticket_keywords
FROM ticket
```

get, composite PKs

```
>>> t = session.query(Ticket).get(1)
>>> t
Ticket(1, 'document arguments for table, column, ...')
>>> print type(t).__name__
Ticket
>>> print t.owner
ash
>>> t.changes
[Change(1, 1132972880, 'component', 'documentation'),
 Change(1, 1142525198, 'owner', 'ash'),
 Change(1, 1142525198, 'comment', 'I think ...'), ...]

>>> session.query(Change).get((1, 1132972880, 'component'))
Change(1, 1132972880, 'component', 'documentation')
>>> print type(_).__name__
Change
```

filter_by, all, order_by

```
>>> q1 = session.query(Ticket).filter_by(reporter='ellisj')
>>> q1.all()
[Ticket(260, 'insert, update have unused **kwargs'),
 Ticket(1061, 'easy_install-2.3 installs ...')]

>>> q1.order_by(desc(Ticket.changetime)).all()
[Ticket(1061, 'easy_install-2.3 installs ...'),
 Ticket(260, 'insert, update have unused **kwargs')]
```

first, one, slicing

```
>>> q2 = session.query(Ticket) \
          .filter_by(reporter='zzzeek',
                     priority='highest') \
          .order_by(Ticket.time)
>>> q2.first()
Ticket(185, 'join object is a little stupid ...')

>>> q2.one()
Traceback (most recent call last):
...
MultipleResultsFound: Multiple rows were found for one()

>>> q2.limit(3).offset(10).all()
[Ticket(385, 'mappers/relations need ...'),
 Ticket(388, 'dirty objects ...'),
 Ticket(389, 'performance patch for attributes')]

>>> q2[10:13] == _
True
```

filter, &, and_

```
>>> session.query(Ticket) \
    .filter((Ticket.reporter.like('zzz%'))
            & (Ticket.summary > 'a')) \
    .first()
Ticket(1, 'document arguments for table, column, ...')
>>> session.query(Ticket) \
    .filter(and_(Ticket.reporter.like('zzz%'),
                 Ticket.summary > 'a')) \
    .first()
Ticket(1, 'document arguments for table, column, ...')
```

Questions?

eagerload, defer

```
>>> print session.query(Ticket).options(eagerload('changes'))
SELECT ticket.id AS ticket_id, ...
       ticket_change_1.ticket AS ticket_change_1_ticket, ...
FROM ticket LEFT OUTER JOIN ticket_change AS ticket_change_1
  ON ticket.id = ticket_change_1.ticket

>>> session.query(Ticket).options(defer('description'))
<sqlalchemy.orm.query.Query object at ...>
```

filter by object

```
>>> t1 = session.query(Ticket).first()
>>> session.query(Change) \
    .filter_by(ticket=t1, field='resolution') \
    .one()
Change(1, 1149120761, 'resolution', 'fixed')
```

any, has

```
>>> print session.query(Ticket).filter(Ticket.changes.any())
SELECT ticket.id AS ticket_id, ...
FROM ticket
WHERE EXISTS
    (SELECT 1
     FROM ticket_change
     WHERE ticket.id = ticket_change.ticket)
```

```
>>> print session.query(Change) \
    .filter(Change.ticket.has(reporter='ellisj'))
SELECT ticket_change.ticket AS ticket_change_ticket, ...
FROM ticket_change
WHERE EXISTS
    (SELECT 1
     FROM ticket
     WHERE ticket.id = ticket_change.ticket
       AND ticket.reporter = ?
```

composable

```
>>> clause = ((Ticket.reporter=='ellisj')
              | (~Ticket.reporter.in_(['zzzeek', 'jek'])))
>>> print session.query(Change) \
    .filter(Change.ticket.has(clause))
SELECT ticket_change.ticket AS ticket_change_ticket, ...
FROM ticket_change
WHERE EXISTS (SELECT 1
FROM ticket
WHERE ticket.id = ticket_change.ticket
  AND (ticket.reporter = ?
       OR ticket.reporter NOT IN (?, ?)))
```

Questions?

from_statement

```
>>> sql = r'''
... SELECT *
...   FROM ticket
...  WHERE summary ~ '[[:digit:]]+'
... '''
>>> session.query(Ticket).from_statement(sql).first()
Ticket(5, 'add support for psycopg1 mxdatetime conversinos')
```

selecting multiple types

```
>>> sql = r'''
...    SELECT *
...      FROM ticket JOIN ticket_change
...           ON (ticket.id = ticket_change.ticket)
...     WHERE summary ~ '[[:digit:]]+'
... ORDER BY ticket.time desc
...    LIMIT 1
... '''
>>> session.query(Ticket, Change).from_statement(sql).first()
Traceback (most recent call last):
...
InvalidRequestError: Ambiguous column name 'time'
in result set! try 'use_labels' option on select statement.
```

working multiple types

```
>>> q3 = session.query(Ticket, Change) \
          .select_from(tickets.join(ticket_changes)) \
          .filter(Ticket.id==1) \
          .order_by(desc(Change.time))
>>> print q3
SELECT ticket.id AS ticket_id, ...
       ticket_change.ticket AS ticket_change_ticket, ...
FROM ticket JOIN ticket_change
     ON ticket.id = ticket_change.ticket
WHERE ticket.id = %(id_1)s
ORDER BY ticket_change.time DESC
>>> q3.first()
(Ticket(1, 'document arguments for table, column, ...'),
 Change(1, 1149120761, 'status', 'closed'))
```

a type and a scalar

```
>>> subquery = select(['count(*)'],
...                   "ticket_change.ticket = ticket.id",
...                   from_obj=ticket_changes)
>>> q4 = session.query(Ticket, subquery.as_scalar())
>>> print q4
SELECT ticket.id AS ticket_id, ...,
       (SELECT count(*)
        FROM ticket_change
        WHERE ticket_change.ticket = ticket.id) AS anon_1
FROM ticket
>>> q4.first()
(Ticket(1, 'document arguments for table, column, ...'),
 7)
```

Questions?

SQLAlchemy

www.sqlalchemy.org