SQLAlchemy 0.4 and Beyond

A trip through the past

SQLAlchemy 0.1

SQLAlchemy 0.1

- Released Feb 13, 2006

SQLAlchemy 0.1

- Released Feb 13, 2006
- API was a fraction of the current: `Table`, `Column`, `engine`, `mapper` and `objectstore.commit()`

SQLAlchemy 0.1

- Released Feb 13, 2006
- API was a fraction of the current: `Table`, `Column`, `engine`, `mapper` and `objectstore.commit()`
- Threadlocal everything, everywhere, always

SQLAlchemy 0.1

- Released Feb 13, 2006
- API was a fraction of the current: `Table`, `Column`, `engine`, `mapper` and `objectstore.commit()`
- Threadlocal everything, everywhere, always
- An alternative, SQLObject-like interface was planned at first, but abandoned to focus on just one idea to start

SQLAlchemy 0.1

- Released Feb 13, 2006
- API was a fraction of the current: `Table`, `Column`, `engine`, `mapper` and `objectstore.commit()`
- Threadlocal everything, everywhere, always
- An alternative, SQLObject-like interface was planned at first, but abandoned to focus on just one idea to start
- ActiveMapper quickly invented by others to provide activerecord-like interface (as well as SQLSoup)

SQLAlchemy 0.2

SQLAlchemy 0.2

- Here come the users !

SQLAlchemy 0.2

- Here come the users !
- Needed to change the API of 0.1, radically, quickly

SQLAlchemy 0.2

- Here come the users !
- Needed to change the API of 0.1, radically, quickly
- Introduced the "decoupled" API: `MetaData`, `Session`, `Query`, `Connection`, `Transaction`, `Dialect`, etc., etc.

SQLAlchemy 0.2

- Here come the users !
- Needed to change the API of 0.1, radically, quickly
- Introduced the "decoupled" API: `MetaData`, `Session`, `Query`, `Connection`, `Transaction`, `Dialect`, etc., etc.
- Threadlocal everything turned off, but all available as extensions/options

SQLAlchemy 0.2

- Here come the users !
- Needed to change the API of 0.1, radically, quickly
- Introduced the "decoupled" API: `MetaData`, `Session`, `Query`, `Connection`, `Transaction`, `Dialect`, etc., etc.
- Threadlocal everything turned off, but all available as extensions/options
- Written in a *super* hurry, didn't want more 0.1-style apps written, wanted to get the most "future" out of one big upgrade while userbase was still small

SQLAlchemy 0.3

SQLAlchemy 0.3

- Thanks for sticking it out through 0.2 ! We're better off.

SQLAlchemy 0.3

- Thanks for sticking it out through 0.2 ! We're better off.
- Mostly focused on rewriting of internals; upgrade experience was mostly unnoticeable

SQLAlchemy 0.3

- Thanks for sticking it out through 0.2 ! We're better off.
- Mostly focused on rewriting of internals; upgrade experience was mostly unnoticeable
- ORM and engine internals reorganized and highly refactored

SQLAlchemy 0.3

- Thanks for sticking it out through 0.2 ! We're better off.
- Mostly focused on rewriting of internals; upgrade experience was mostly unnoticeable
- ORM and engine internals reorganized and highly refactored
- User needs continued to be evaluated; made room for polymorphic class loading, better connection pool behavior, smarter checking of state upon flush

SQLAlchemy 0.3

- Thanks for sticking it out through 0.2 ! We're better off.
- Mostly focused on rewriting of internals; upgrade experience was mostly unnoticeable
- ORM and engine internals reorganized and highly refactored
- User needs continued to be evaluated; made room for polymorphic class loading, better connection pool behavior, smarter checking of state upon flush
- People were now building things, API had stabilized, things worked "OK"

SQLAlchemy 0.4

SQLAlchemy 0.4

- Here come the developers !

SQLAlchemy 0.4

- Here come the developers !
- Like 0.3, was oriented around major internal refactorings

SQLAlchemy 0.4

- Here come the developers !
- Like 0.3, was oriented around major internal refactorings
- Speed profiling in earnest began; nearly all remnants of earlier versions reworked

SQLAlchemy 0.4

- Here come the developers !
- Like 0.3, was oriented around major internal refactorings
- Speed profiling in earnest began; nearly all remnants of earlier versions reworked
- SQL expression constructs became simpler, quicker, more sophisticated

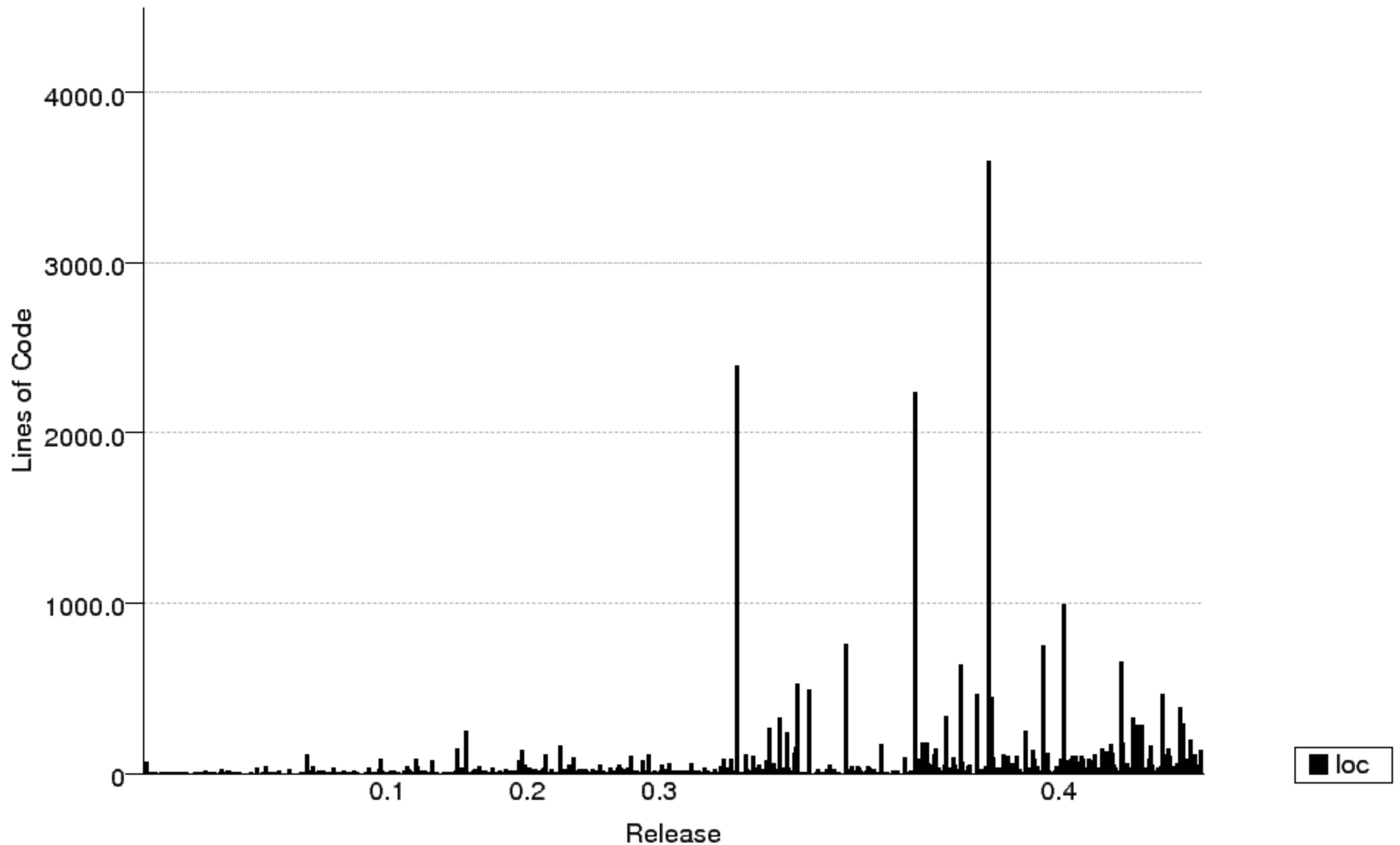
SQLAlchemy 0.4

- Here come the developers !
- Like 0.3, was oriented around major internal refactorings
- Speed profiling in earnest began; nearly all remnants of earlier versions reworked
- SQL expression constructs became simpler, quicker, more sophisticated
- transactional system reworked, support for two phase and SAVEPOINT added

SQLAlchemy 0.4

- Here come the developers !
- Like 0.3, was oriented around major internal refactorings
- Speed profiling in earnest began; nearly all remnants of earlier versions reworked
- SQL expression constructs became simpler, quicker, more sophisticated
- transactional system reworked, support for two phase and SAVEPOINT added
- Query object pushed into the next level; virtually every pattern in place at the beginning of 0.3 was deprecated

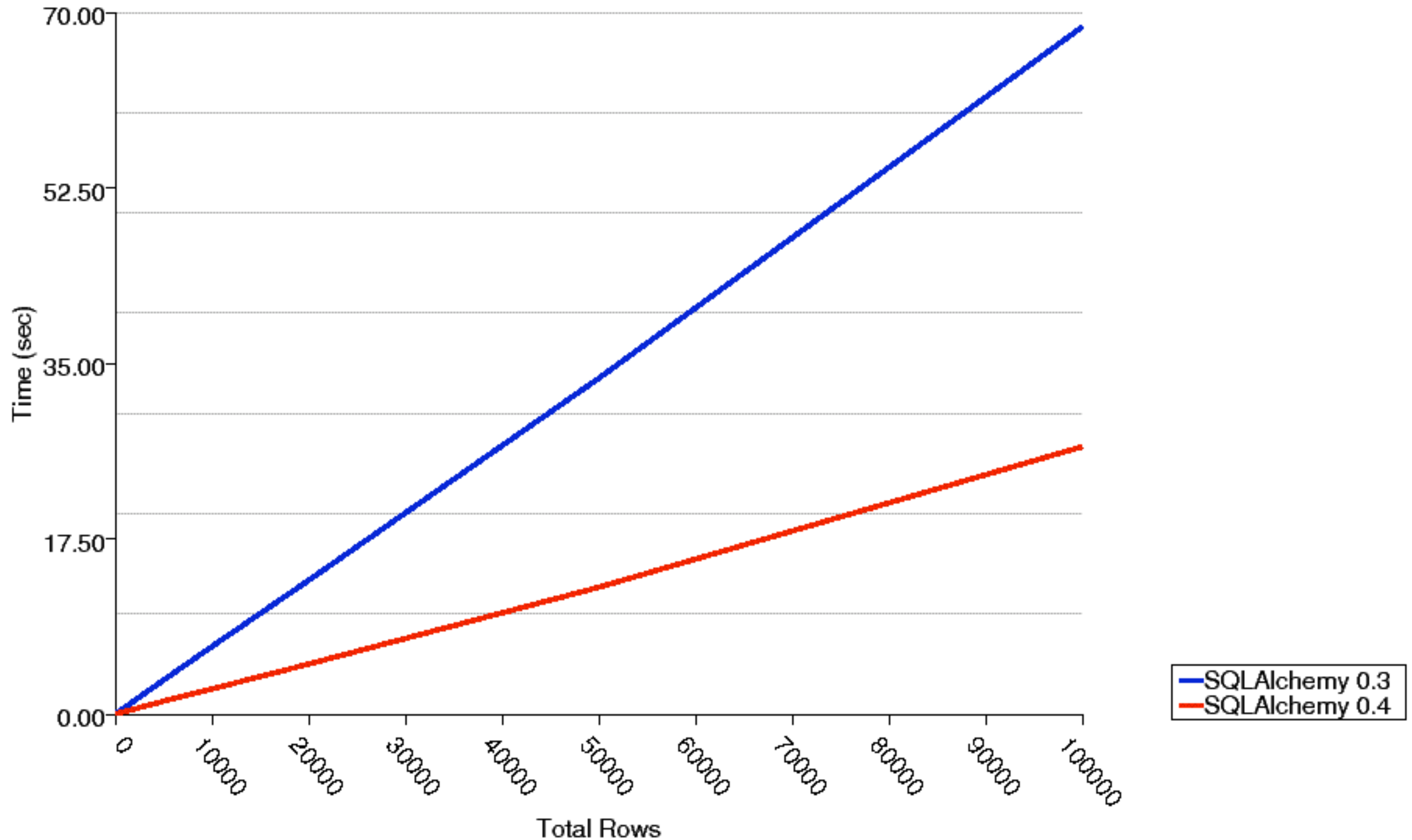
How Much of 0.4 is New ?



Highlights of 0.4

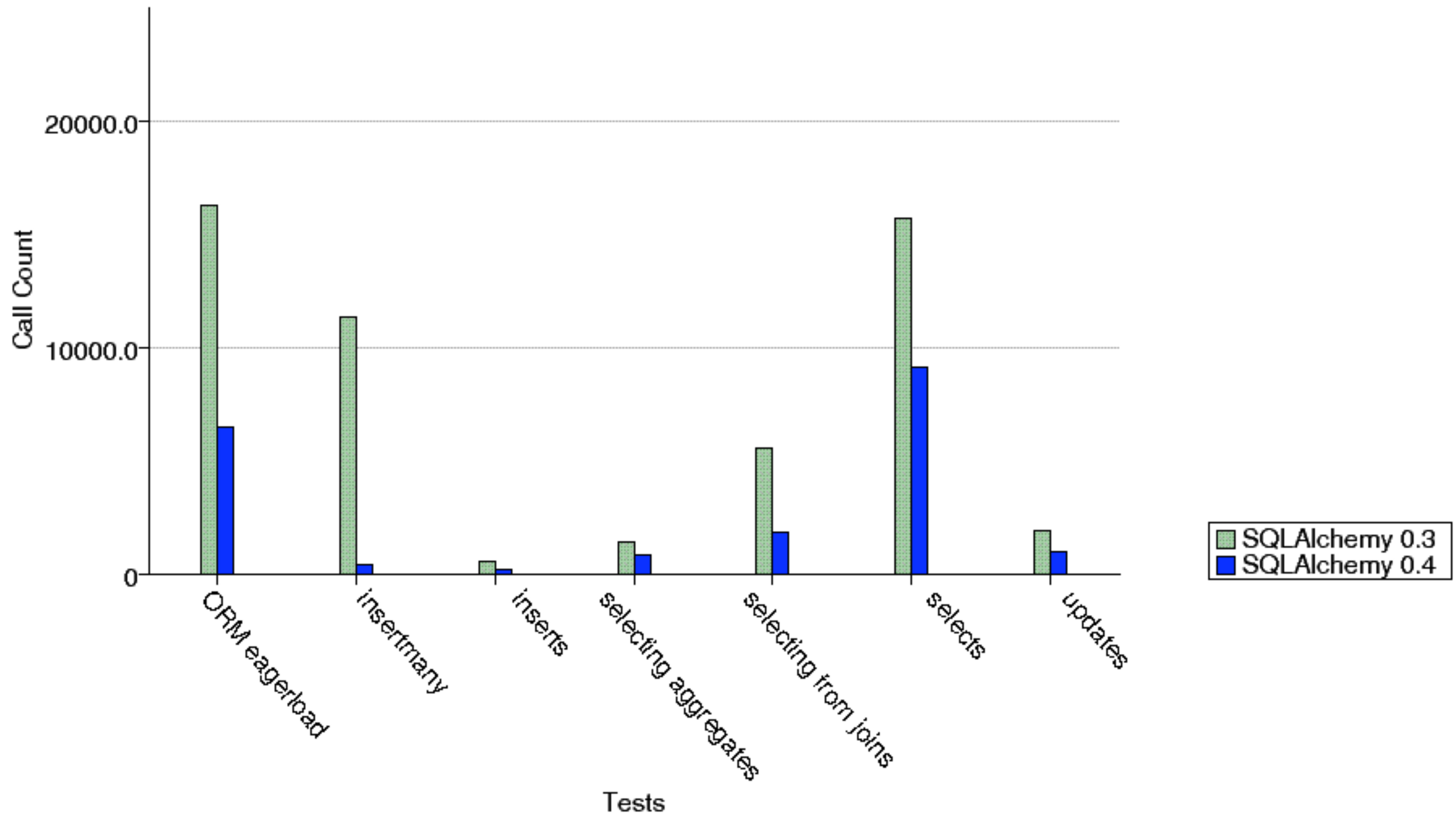
Speed !

We fixed a few honkers



Speed !

..and reduced complexity and/or callcounts in all areas



SQL Expression Language

Generative SQL Expressions

Selects and CRUD expressions are constructed across multiple method calls.

```
s = select([employees.c.id, employees.c.name])

s = s.where(employees.c.id.between(250, 300))

s = s.where(employees.c.name.like('%smith%'))

s = s.order_by(employees.c.dept)

for row in s.execute():
    print row
```

Smart Operators

Operators take datatypes and underlying database dialect into account when rendered.

```
>>> sometable = Table('sometable',
...     metadata,
...     Column('intvalue', Integer),
...     Column('stringvalue', String)
... )

>>> print sometable.c.intvalue + 3

sometable.intvalue + :intvalue
```

Smart Operators

Operators take datatypes and underlying database dialect into account when rendered.

```
>>> sometable = Table('sometable',
...     metadata,
...     Column('intvalue', Integer),
...     Column('stringvalue', String)
... )

>>> print sometable.c.stringvalue + 'hi'

sometable.stringvalue || :stringvalue
```

Smart Operators

Operators take datatypes and underlying database dialect into account when rendered.

```
>>> sometable = Table('sometable',
...     metadata,
...     Column('intvalue', Integer),
...     Column('stringvalue', String)
... )

>>> metadata.bind = \
      create_engine('mysql://')
>>> print sometable.c.stringvalue + 'hi'
```

Smart Operators

Operators take datatypes and underlying database dialect into account when rendered.

```
>>> sometable = Table('sometable',
...     metadata,
...     Column('intvalue', Integer),
...     Column('stringvalue', String)
... )

>>> metadata.bind = \
         create_engine('mysql://')
>>> print sometable.c.stringvalue + 'hi'

concat(sometable.stringvalue, :stringvalue)
```

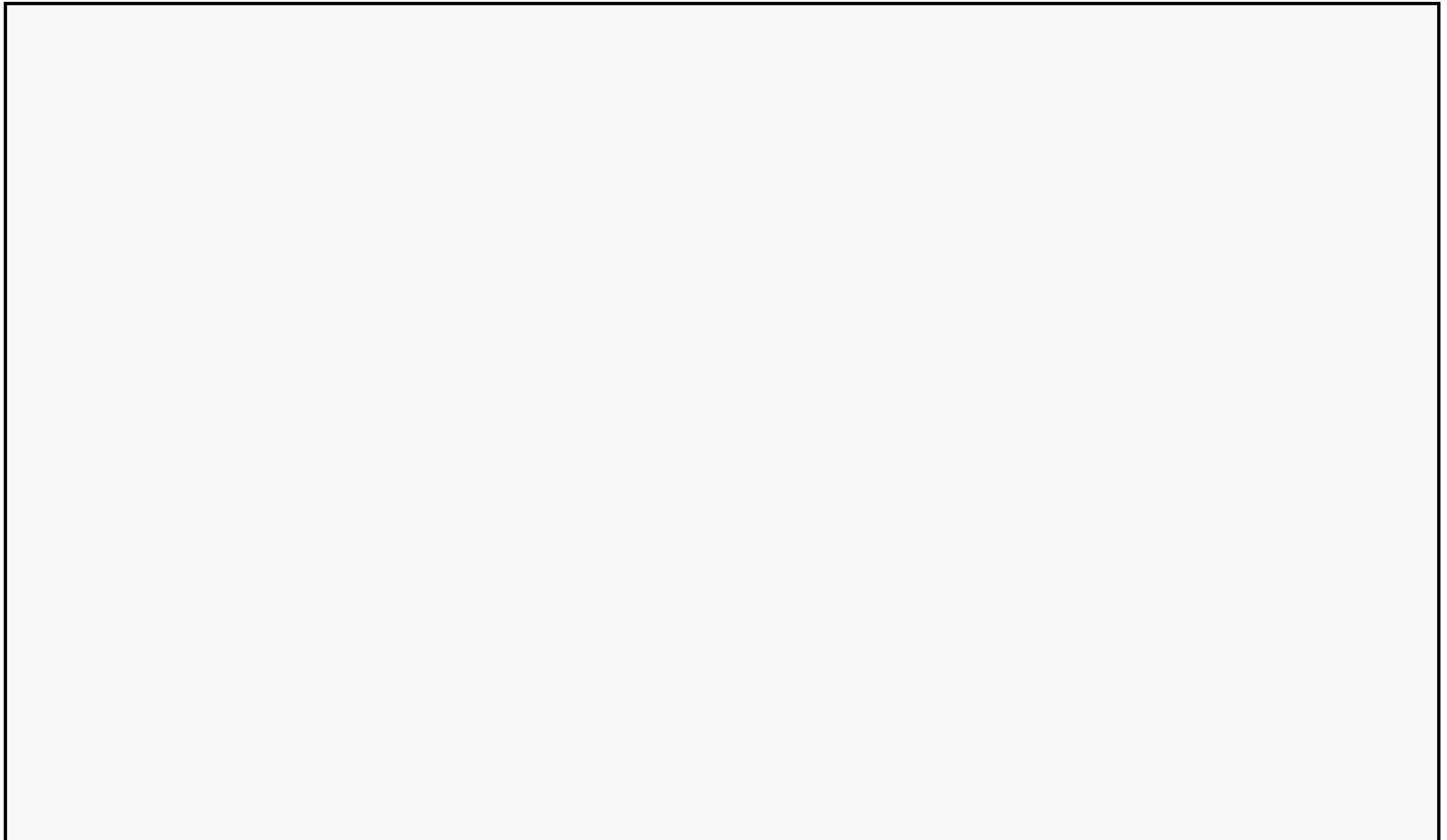
Smart Operators

Operators take datatypes and underlying database dialect into account when rendered.

```
>>> sometable = Table('sometable',
...     metadata,
...     Column('intvalue', Integer),
...     Column('stringvalue', String)
... )

>>> metadata.bind = \
        create_engine('mysql://')
>>> print sometable.c.stringvalue + 'hi'

concat(sometable.stringvalue, :stringvalue)
```

Generic Functions

Known SQL function calls are datatype-aware and adjust to underlying database dialect



Generic Functions

Known SQL function calls are datatype-aware and adjust to underlying database dialect

```
>>> s = select([func.now()])
>>> print s
SELECT now() AS now_1
```

Generic Functions

Known SQL function calls are datatype-aware and adjust to underlying database dialect

```
>>> s = select([func.now()])
>>> print s
SELECT now() AS now_1

>>> s.bind = create_engine('sqlite://')
>>> print s
SELECT CURRENT_TIMESTAMP AS now_1
```

Generic Functions

Known SQL function calls are datatype-aware and adjust to underlying database dialect

```
>>> s = select([func.now()])
>>> print s
SELECT now() AS now_1

>>> s.bind = create_engine('sqlite://')
>>> print s
SELECT CURRENT_TIMESTAMP AS now_1

>>> print func.now().type
DateTime(timezone=False)
```

New ORM Query API

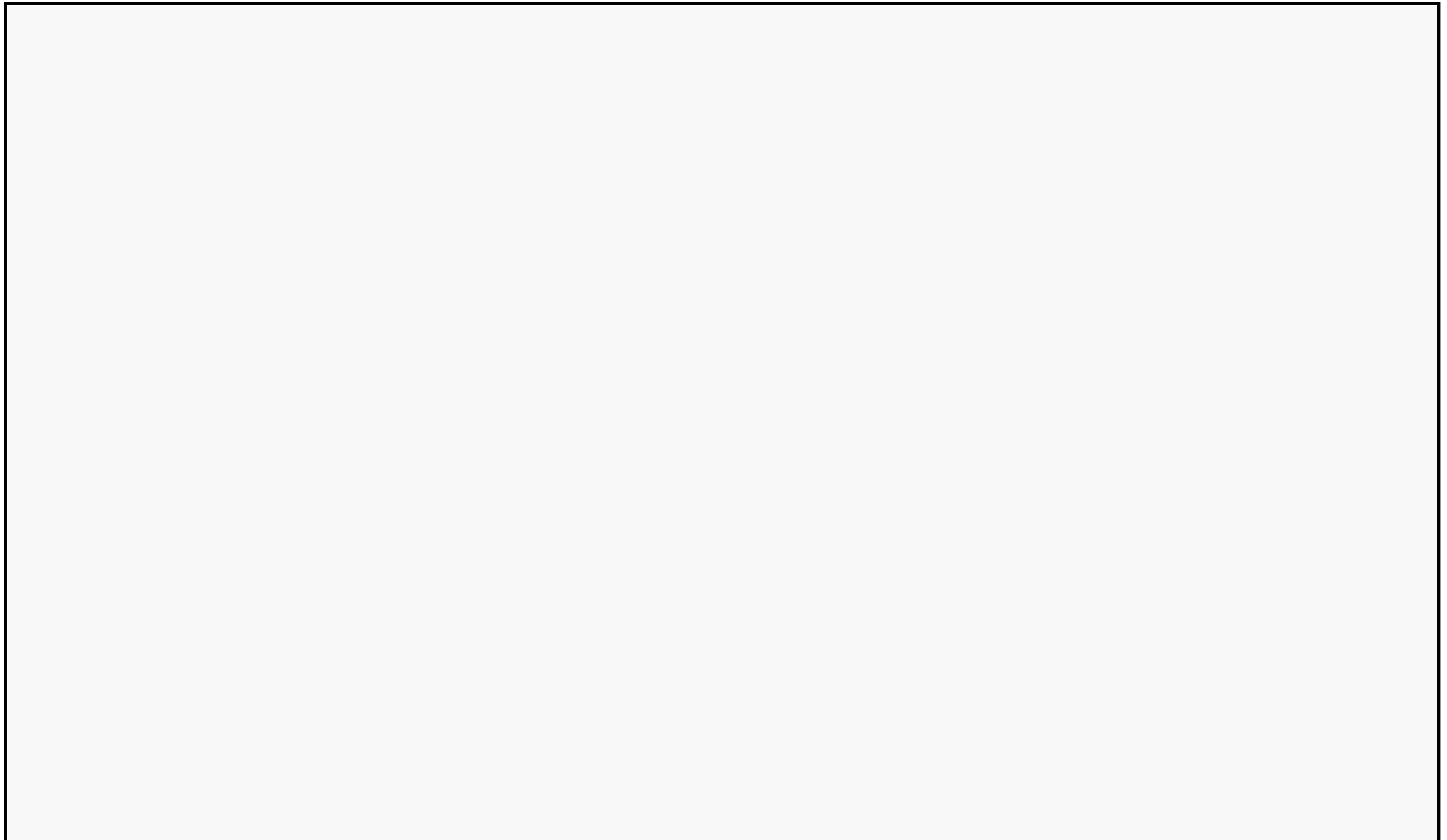
Generative Queries

Query object is fully generative. Mapped properties now have relational operators.

```
query = session.query(User)

query = query.filter(User.name == 'jack')

query = query.order_by(User.lastname)

for user in query:
    print user.lastname
```

Inline Aliasing

Table aliases can be generated and filtered against within a single query expression



Inline Aliasing

Table aliases can be generated and filtered against within a single query expression

```
>>> categories = Table('categories',
...     metadata,
...     Column('id', String,
...            primary_key=True)
...     Column('name', String),
...     Column('subcategory_of', String,
...        ForeignKey('categories.id'))
...     )
```

Inline Aliasing

Table aliases can be generated and filtered against within a single query expression

```
>>> class Category(object):
...     pass

>>> mapper(Category, categories,
...  properties={
...    'parent_category':relation(Category,
...         remote_side=categories.c.id,
...         backref='subcategories')
...    })
```

Inline Aliasing

In version 0.3, finding subcategories of a parent category required manual aliasing:

```
>>> calias = categories.alias()

>>> session.query(Category).\
...   select_from(
...      categories.join(calias,
...            calias.c.id==
...     categories.c.subcategory_of)).\
...   filter(calias.c.name=='somecategory')
```

Inline Aliasing

In 0.4, join() generates aliased joins and aliases criterion inline, so it's just:

```
>>> session.query(Category).\
...     join('parent_category', aliased=True).\
...     filter(Category.name=='somecategory')
```

Inline Aliasing

In 0.4, join() generates aliased joins and aliases criterion inline, so it's just:

```
>>> session.query(Category).\
...     join('parent_category', aliased=True).\
...     filter(Category.name=='somecategory')

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of
FROM categories JOIN categories AS
categories_1 ON categories_1.id =
categories.subcategory_of
WHERE categories_1.name = ? ORDER BY
categories.oid
```

Inline Aliasing

Example: search for the path "USA/Illinois/Chicago"

```
>>> session.query(Category).\
...   filter(Category.name=='Chicago').\
...   join('parent_category',aliased=True).\
...   filter(Category.name=='Illinois').\
...   join('parent_category',
...     aliased=True, from_joinpoint=True).\
...   filter(Category.name=='USA').all()
```

Inline Aliasing

Example: search for the "USA/Illinois/Chicago"

```
SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of
FROM categories JOIN categories AS
categories_1 ON categories_1.id =
categories.subcategory_of JOIN categories
AS categories_2 ON categories_2.id =
categories_1.subcategory_of
WHERE categories.name = ? AND
categories_1.name = ? AND categories_2.name
= ? ORDER BY categories.oid
['Chicago', 'Illinois', 'USA']
```

Higher Level Operators

New operators such as "has()" and "any()" intelligently generate inter-table criteria

```
>>> session.query(Category).filter(
...     Category.parent_category.has(
...         Category.name=='USA'
...     )
... ).all()
```

Higher Level Operators

New operators such as "has()" and "any()" intelligently generate inter-table criteria

```
>>> session.query(Category).filter(
...     Category.parent_category.has(
...         Category.name=='USA'
...     )
... ).all()

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of FROM categories
WHERE EXISTS (SELECT 1 FROM categories AS
categories_1 WHERE categories_1.id =
categories.subcategory_of AND categories_1.name
= ?) ORDER BY categories.oid
['USA']
```

Higher Level Operators

New operators such as "has()" and "any()" intelligently generate inter-table criteria

```
>>> session.query(Category).filter(
...       ~Category.subcategories.any()
... ).all()
```

Higher Level Operators

New operators such as "has()" and "any()" intelligently generate inter-table criteria

```
>>> session.query(Category).filter(
...     ~Category.subcategories.any()
... ).all()

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of FROM categories
WHERE NOT (EXISTS (SELECT 1 FROM categories
AS categories_1 WHERE categories.id =
categories_1.subcategory_of)) ORDER BY
categories.oid
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...   filter(Category.name=='Chicago').\
...   one()
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...     filter(Category.name=='Chicago').\
...     one()

>>> session.query(Category).\
...    filter(
...      Category.parent_category==chicago)
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...   filter(Category.name=='Chicago').\
...   one()

>>> session.query(Category).\
...  filter(
...    Category.parent_category==chicago)

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of FROM categories
WHERE ? = categories.subcategory_of ORDER BY
categories.oid
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...     filter(Category.name=='Chicago').\
...     one()
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...     filter(Category.name=='Chicago').\
...     one()

>>> session.query(Category).\
...    filter(Category.subcategories.\
...           contains(chicago))
```

Higher Level Operators

Object instances usable in criterion

```
>>> chicago = session.query(Category).\
...     filter(Category.name=='Chicago').\
...     one()

>>> session.query(Category).\
...    filter(Category.subcategories.\
...            contains(chicago))

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of FROM categories
WHERE categories.id = ? ORDER BY
categories.oid
```

New ORM Configurations

New "Declarative" Layer

Familiar Table and mapper constructs can be moved into class declarations

```
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column('id', Integer, primary_key=True)
    name = Column('name', String(50))
    addresses = relation("Address", backref="user")

class Address(Base):
    __tablename__ = 'addresses'

    id = Column('id', Integer, primary_key=True)
    email = Column('email', String(50))
    user_id = Column('user_id', Integer,
       ForeignKey('users.id'))
```

Collections API

Sets, dictionaries, and any user-defined collection may be mapped using an open API

```
mapper(MyClass, sometable, properties={
    'elements':relation(Element,
        collection_class=set)
})
```

Collections API

Sets, dictionaries, and any user-defined collection may be mapped using an open API

```
from sqlalchemy.orm.collections \
   import collection

class MySet(object):
    @collection.appender
    def add_element(self, elem):
        # ...

mapper(MyClass, sometable, properties={
   'elements':relation(Element,
      collection_class=MySet)
})
```

Collections API

Several "pre-fab" collections are available for creating attribute/column-mapped dictionaries, etc.

```
from sqlalchemy.collections import \
   attribute_mapped_collection

mapper(MyClass, sometable, properties={
   'elements':relation(Element,
      collection_class=
      attribute_mapped_collection(
          'element_name'
      )
})
```

"Dynamic" Relations

Very large collections can be managed by a "dynamic" relation, which issues queries for every access

```
mapper(Category, categories, properties={
   'subcategories':relation(Category,
             lazy="dynamic")
})
```

"Dynamic" Relations

Very large collections can be managed by a "dynamic" relation, which issues queries for every access

```
>>> for subcat in cat.subcategories[5:10]:
...     print subcat
```

"Dynamic" Relations

Very large collections can be managed by a "dynamic" relation, which issues queries for every access

```
>>> for subcat in cat.subcategories[5:10]:
...     print subcat

SELECT categories.id AS categories_id,
categories.name AS categories_name,
categories.subcategory_of AS
categories_subcategory_of FROM categories
WHERE ? = categories.subcategory_of ORDER
BY categories.oid  LIMIT 5 OFFSET 5
[5]
```

"Dynamic" Relations

Dynamic relations are still writable, and keep track of appends/deletes

```
>>> cat.subcategories.remove(somecat)
>>> cat.subcategories.append(someothercat)
>>> session.flush()
```

"Dynamic" Relations

Dynamic relations are still writable, and keep track of appends/deletes

```
>>> cat.subcategories.remove(somecat)
>>> cat.subcategories.append(someothercat)
>>> session.flush()

BEGIN
UPDATE categories SET subcategory_of=?
WHERE categories.id = ?
[None, 5]
INSERT INTO categories (name,
subcategory_of) VALUES (?, ?)
['subcat', 1]
COMMIT
```

Polymorphic Inheritance

Whole inheritance hierarchies can be loaded automatically

```
employees= Table('employees', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('type', String))

engineers= Table('engineers', metadata,
    Column('id', Integer,
ForeignKey('employees.id'), primary_key=True),
    Column('status', String))

managers = Table('managers', metadata,
    Column('id', Integer,
ForeignKey('employees.id'), primary_key=True),
    Column('title', String))
```

Polymorphic Inheritance

Whole inheritance hierarchies can be loaded automatically

```
class Employee(object):pass
class Manager(Employee):pass
class Engineer(Employee):pass

mapper(Employee, employees,
    polymorphic_on=employees.c.type)

mapper(Manager, managers, inherits=Employee,
    polymorphic_identity='manager')

mapper(Engineer, engineers, inherits=Employee,
    polymorphic_identity='engineer')
```

Polymorphic Inheritance

Whole inheritance hierarchies can be loaded automatically

```
>>> session.query(Employee).all()
```

Polymorphic Inheritance

Whole inheritance hierarchies can be loaded automatically

```
>>> session.query(Employee).all()

SELECT employees.id AS employees_id,
employees.name AS employees_name, employees.type
AS employees_type FROM employees ORDER BY
employees.oid
[]
SELECT managers.id AS managers_id, managers.title
AS managers_title FROM managers WHERE ? =
managers.id
[1]
SELECT engineers.id AS engineers_id,
engineers.status AS engineers_status FROM
engineers WHERE ? = engineers.id
[2]
```

Polymorphic Inheritance

with_polymorphic() can generate joins on the fly to reduce queries...

```
>>> session.query(Employee).\
...     with_polymorphic('*').all()
```

Polymorphic Inheritance

with_polymorphic() can generate joins on the fly to reduce queries...

```
>>> session.query(Employee).\
...     with_polymorphic('*').all()

SELECT employees.name AS employees_name,
employees.id AS employees_id, managers.id AS
managers_id, managers.title AS managers_title,
engineers.id AS engineers_id, engineers.status AS
engineers_status, employees.type AS employees_type
FROM employees LEFT OUTER JOIN managers ON
employees.id = managers.id LEFT OUTER JOIN
engineers ON employees.id = engineers.id ORDER BY
employees.oid
[]
```

Polymorphic Inheritance

...or to specify criteria against a subclass

```
>>> session.query(Employee).\
...     with_polymorphic(Engineer).\
...     filter(or_(
...         Employee.name.like('%e%'),
...         Engineer.status != 'EE'))
```

Polymorphic Inheritance

...or to specify criteria against a subclass

```
>>> session.query(Employee).\
...   with_polymorphic(Engineer).\
...   filter(or_(
...      Employee.name.like('%e%'),
...      Engineer.status != 'EE'))
```

```
SELECT employees.id AS employees_id, employees.name
AS employees_name, engineers.id AS engineers_id,
engineers.status AS engineers_status,
employees.type AS employees_type FROM employees
LEFT OUTER JOIN engineers ON employees.id =
engineers.id
WHERE employees.name LIKE ? OR
engineers.status != ? ORDER BY employees.oid
['%e%', 'EE']
```

Polymorphic Inheritance

The of_type() operator serves the function of with_polymorphic() when querying on relations

```
mapper(Company, companies, properties={
    'employees':relation(Employee)
})
```

Polymorphic Inheritance

The of_type() operator serves the function of with_polymorphic() when querying on relations

```
mapper(Company, companies, properties={
    'employees':relation(Employee)
})

>>> session.query(Company).\
...   filter(
...    Company.employees.of_type(Engineer).\
...    any(Engineer.status == 'EE')
...   ).all()
```

New Transactional Features

Transactional Sessions

Sessions can be configured to be "always transactional", and optionally auto-flushing

```
>>> Session = sessionmaker(
...     transactional=True, autoflush=True)

>>> sess = Session()
>>> u = sess.query(User).get(5)
>>> u.name = 'new name'

>>> u2 = sess.query(User).\
...      filter(User.name=='new name').one()
>>> sess.commit()
```

Transaction Nesting

Databases which support SAVEPOINT can nest transactions, allowing rollback-to-savepoint behavior

```
>>> conn = engine.connect()
>>> trans = conn.begin()
>>> conn.execute(users.insert(), name='ed')

>>> nested = conn.begin_nested()
>>> conn.execute(users.insert(), name='jack')
>>> nested.rollback()

>>> trans.commit()
```

Two Phase Commit

Connections and ORM Sessions both support two-phase commit semantics for supported databases

```
>>> trans1 = conn1.begin_twophase()
>>> trans2 = conn2.begin_twophase()

>>> conn1.execute(users.insert(), name='ed')
>>> conn2.execute(logreg.insert(), value='ins')

>>> try:
...     trans1.prepare()
...     trans2.prepare()
...     trans1.commit()
...     trans2.commit()
... except:
...     trans1.rollback()
...     trans2.rollback()
...
```

And More !

Other Features

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior
- metadata.reflect() can load full schemas at once

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior
- metadata.reflect() can load full schemas at once
- New dialects: MaxDB, Sybase, Access, Informix, DB2

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior
- metadata.reflect() can load full schemas at once
- New dialects: MaxDB, Sybase, Access, Informix, DB2
- "horizontal sharding" extension for ORM; transparently loads and saves rows across multiple databases

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior
- metadata.reflect() can load full schemas at once
- New dialects: MaxDB, Sybase, Access, Informix, DB2
- "horizontal sharding" extension for ORM; transparently loads and saves rows across multiple databases
- "connection events" API allows configuration of per-connection / per-use actions on connections, such as isolation mode adjustments, etc.

Other Features

- ORM supports mutable primary keys, ON UPDATE cascade
- Arbitrary SQL expressions can be assigned to object attributes for "atomic" update behavior
- metadata.reflect() can load full schemas at once
- New dialects: MaxDB, Sybase, Access, Informix, DB2
- "horizontal sharding" extension for ORM; transparently loads and saves rows across multiple databases
- "connection events" API allows configuration of per-connection / per-use actions on connections, such as isolation mode adjustments, etc.

What's Coming Up

What's Coming Up

- Migrate is back !

What's Coming Up

- Migrate is back !
- Dialects will soon decouple SQL compilers from DBAPI behaviors - will allow easy reuse among JDBC, ODBC, multiple native DBAPI connectors

What's Coming Up

- Migrate is back !
- Dialects will soon decouple SQL compilers from DBAPI behaviors - will allow easy reuse among JDBC, ODBC, multiple native DBAPI connectors
- Jython support !

What's Coming Up

- Migrate is back !
- Dialects will soon decouple SQL compilers from DBAPI behaviors - will allow easy reuse among JDBC, ODBC, multiple native DBAPI connectors
- Jython support !
- Customizable class instrumentation; PJE using it to integrate with Trellis

What's Coming Up

- Migrate is back !
- Dialects will soon decouple SQL compilers from DBAPI behaviors - will allow easy reuse among JDBC, ODBC, multiple native DBAPI connectors
- Jython support !
- Customizable class instrumentation; PJE using it to integrate with Trellis
- SQLAlchemy ~~Book~~ Books

Fin !

http://www.sqlalchemy.org

http://techspot.zzzeek.org