PostgreSQL, SQLAlchemy, and schema-less data.
MySQL is the root of all evil

- Most developers met 'databases' through MySQL.
  - MySQL is a terrible, limited feature, SQL-oriented database.
  - Developers believed MySQL was “typical” and defined SQL-database-ness.
    - It was not, is not, and does not.
  - MySQL's, not SQL's, deficiencies led to the creation of the “noSQL” category of data-store solutions.
Lethe

Let us forget about MySQL.

Take a moment.

Cleanse your mind.
noSQL Is Not About SQL

• noSQL is a false category
  – It is not about SQL.
  – The distinction is between schema-less and schema-enforce data-stores.
  – Nothing about SQL requires a schema.

• “Traditional” data-stores typically use the term “record” while hipster data-stores use the term “document”
  • This is a distinction without a difference.
  • What if your “record” could contain anything?
    – It would be a document.
SQL vs. Map-Reduce / 'sharding'

• A distinction without much of a difference.
  - 'Traditional' databases have supported some forms of parallelism for years.
    • There are devils in these details, as expected.
  - Informix PDQ “Parallel Data Query” compiles SQL queries into parallel execution paths and aggregates the result.
    • That feature first introduced in Informix IDS 7.10
      - December 1994!
  - A modern database query optimizer...
    • ... is probably smarter than you.
Challenges

• The most significant challenge for schema-less data:
  – Indexing
    • How?
  – Representation
    • Native representation is bloated [JSON or XML]
      – Irregular sizes result in inefficient I/O
    • Difficult to journalize changes [transactions]
    • 'Upgrades' to the schema-less data that, honestly, actually has a schema.
      – Ok, not a schema, but 'expectations'.
Options (in PostgreSQL)

- **XML**
  - Expression indexing
  - XPath support & schema validation.

- **JSON**
  - V8 JavaScript engine can be embedded in PG
  - Expression indexing

- **HSTORE**
  - GiST, GIN, & B-Tree indexing
    - Entire hierarchy is indexed
  - Expression indexing
  - Binary internal representation!!!
CREATE TABLE user_data_xml (  
  record_id SERIAL,  
  login VARCHAR(12),  
  user_prefs XML);

CREATE INDEX user_data_xml_tz  
  ON user_data_xml  
  USING btree  
  (((xpath('/Preferences/TimeZone[1]/text()',  
    user_prefs)))[1]::text));

SELECT *  
FROM user_data_xml  
WHERE (xpath('/Preferences/TimeZone[1]/text()',  
  userPrefs))::text='US/Eastern';

---

**Downsides:**
- Non-binary serialization
- You need to know what you will be searching for.

**Upsides**
- XML is flexible
- No encoding issues
- Namespaces
- XPath is powerful.
SELECT *
FROM user_data_json
WHERE user_prefs->>'timezone' = 'US/Eastern';

• Records can be cast to and from JSON, and JSON can be cast to and from HSTORE.
• Array operations supported.

CREATE TABLE user_data_json (record_id SERIAL, login VARCHAR(12), user_prefs JSON);

CREATE INDEX user_data_json_tz ON user_data_json((user_prefs->>'timezone'));

Downsides:
• It is JSON
• You better use UTF-8
• Limited data-types
• You need to know what you will be searching for.
• Non-binary serialization.

Upsides
• It is JSON
JSON SQL Operators

- “->” Get element
- “->>” Get field
- “#>”/”#>>” Array of text
- array_to_json(arr)
- to_json(any)
- json_array_length(x)
- json_each(x)
- row_to_json(r)

SQLAlchemy & JSON

- SQLAlchemy has no direct support for JSON data-type.
  - HSTORE enhancements in PostgreSQL 9.4 would probably render it irrelevant
  - It is relatively simple to add basic JSON support:
    - https://github.com/inklesspen/frameline/blob/master/frameline/models.py
    - but that still lacks operator support.
HSTORE

CREATE EXTENSION hstore;

CREATE TABLE user_data (record_id SERIAL, login VARCHAR(12), prefs HSTORE);

CREATE INDEX user_data_btree ON user_data USING BTREE(prefs);

CREATE INDEX user_data_gin ON user_data USING GIN(prefs);

Downsides:
- It may need to be cast.
- It is not JSON or XML

Upsides
- Full index support.
- Binary serialization.

Nested keys not supported until 9.4
Using HSTORE

INSERT INTO user_data(login, prefs) VALUES('awilliam', 'timezone=>"US/Eastern",zip_code=>"49503"');

SELECT login, prefs FROM user_data;
awilliam | "timezone"=>"US/Eastern", "zip_code"=>"49503"

SELECT * FROM user_data
WHERE prefs->'timezone' = 'US/Eastern';
UPDATE HSTORE Value

UPDATE user_data
SET prefs = prefs || 'busstop=>5941'
WHERE login = 'awilliam';

UPDATE user_data
SET prefs = hstore('outboundBusStop', prefs-'busstop') ||
delete(prefs, 'busstop') || 'inboundBusStop=>5736'
WHERE login = 'awilliam';

SELECT prefs FROM user_data
WHERE prefs-'timezone' = 'US/Eastern';
"timezone"=>'US/Eastern","zip_code"=>'49503',
"outboundBusStop"=>'5941","inboundBusStop"=>'5736"
HSTORE SQL Operators

- “->” Value for key
- “->x[]” Value for keys
- “||” Concatenate
- “?” Contains key
- “?&” Contains all keys
- “?|” Contains any key
- “@>” Contains

- “-” Delete key
- “-x[]” Delete keys
- “x-y” Substack
- “#=” Replace
- “%%” To Array
- “%#” To 2D Array
Indexes

• GiST & GIN
  - Unordered
  - Non-Equality operators: @>, ?, ?& and ?|

• BTREE & HASH
  - Ordered
  - Equality operator: [==]
Indexes Work

EXPLAIN

```
SELECT * FROM user_data
WHERE prefs ? 'timezone';
```

QUERY PLAN

```
Bitmap Heap Scan on user_data
(cost=12.01..16.02 rows=1 width=78)
  Recheck Cond: (prefs ? 'timezone '::text)
  ->  Bitmap Index Scan on user_data_gin
      (cost=0.00..12.01 rows=1 width=0)
      Index Cond: (prefs ? 'timezone '::text)

(4 rows)
```
HSTORE's History

- Created in 2003 (PostgreSQL 7.3)
- Enters PostgreSQL standard (PostgreSQL 8.2, 2006)
- Support for GIN indexes (PostgreSQL 8.3, 2007)
- Limits removed (PostgreSQL 9.0, 2010)
  - previously 64k limit for keys & values
  - records and arrays can be cast to HSTORE type
- Nested Array Support (PostgreSQL 9.4, 2013)
HSTORE Limits

- Elements In Array: $2^{28}$
- Key/Value Pairs: $2^{28}$
- Maximum String Length: $2^{28b}$
- Levels: unlimited
- Length of nested hash/array: $2^{28b}$

So the cap is roughly 256MB
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import create_engine, Column, String, Integer
from sqlalchemy.orm import sessionmaker
from sqlalchemy.dialects.postgresql import HSTORE, ARRAY
from sqlalchemy.ext.mutable import MutableDict

Base = declarative_base()

class User(Base):
    __tablename__ = 'user_data'
    user_id = Column('record_id', Integer, primary_key=True)
    login = Column('login', String(12), nullable=False)
    prefs = Column('prefs', MutableDict.as_mutable(HSTORE))
SQLA's HSTORE Methods

array()
contained_by(other)
contains(other, **kwargs)
defined(key)
delete(key)
has_all(other)
has_any(other)

has_key(other)
keys()
matrix()
slice(array)
vals()
concatenation [+]

Examples

```python
user = session.query(User).\
    filter(User.prefs['timezone']==\'
    'US/Eastern')

user = session.query(User).\
    filter(User.prefs.has_key('inboundBusStop')).one()

user = session.query(User).\
    filter(\n        User.prefs.contains(\n            {'inboundBusStop': u'5736'}\n        )\n    ).one()
```
Updating

user.prefs['timezone'] = 'US/Pacific'
user.prefs['nickname'] = 'whitemice'
del user.prefs['inboundBusStop']
session.commit()

for (x in session.query(User.pref).filter(User.prefs.has_key('inboundBusStop'))):
    del x['inboundBusStop']
session.query(User).
    filter(User.prefs.has_key('inboundBusStop')).
    update(
        {User.prefs: User.prefs + {'transitAvailable': 'true', }, },
        synchronize_session="fetch")

UPDATE user_data SET prefs=(user_data.prefs || %(prefs_1)s)
WHERE user_data.prefs ? %(prefs_2)s
    {'prefs_1': {'transitAvailable': 'true'}, 'prefs_2': 'inboundBusStop'}
session.query(User).\ 
filter(User.prefs.has_key('inboundBusStop')).\ 
update( 
    {User.prefs: User.prefs.delete('inboundBusStop') }, 
    synchronize_session="fetch")

UPDATE user_data SET prefs=delete(user_data.prefs, % (param_1)s) WHERE user_data.prefs ? %%(prefs_1)s 
    {'prefs_1': 'inboundBusStop', 'param_1': 'inboundBusStop'}
Casting To JSON
(server side)

from sqlalchemy import 
    create_engine, Column, String, Integer, func

....

json = session.query(func.hstore_to_json(User.prefs)).
    filter(User.prefs.has_key('inboundBusStop')).first()
print('JSON: {}'.format(json[0], ))

{u'timezone': u'US/Eastern', u'outboundBusStop': u'5941',
  u'inboundBusStop': u'5736', u'zip_code': u'49503'}
Other Related Stuff…
Monges
(Experimental)

- Supports MongoDB's wire-level protocol to a PostgreSQL backend.
  - https://github.com/umitanuki/mongres
  - Requires plv8
    - http://code.google.com/p/plv8js/
  - No license is clearly declared
Mongo_FDW

- PostgreSQL 9.1 officially adds the extensions for Foreign Database Wrappers.
- Version 9.3 added write-through support.
  - Connect to other data-stores using PostgreSQL as a federation engine.
  - MongoDB as a foreign database
    - https://github.com/citusdata/mongo_fdw
Informix/DB2 12.10

“Applications that use the JSON-oriented query language, created by MongoDB, can interact with data stored in Informix® databases. The Informix database server also provides built-in JSON and BSON (binary JSON) data types. You can use MongoDB community drivers to insert, update, and query JSON documents in Informix.”

Complex Data Types

• Most modern databases support complex data types:
  – UUID
  – TEXT (Text is actually a rather complicated thing)
    • Full text search vectors, including linguistic stems.
  – ARRAY
  – CIDR / INET
  – INTERVAL / RANGE

• Not schema-less, but under utilized.