PostgreSQL as a Schemaless Database.

Christophe Pettus
PostgreSQL Experts, Inc.
OSCON 2013
Welcome!

- I’m Christophe.
- PostgreSQL person since 1997.
- Consultant with PostgreSQL Experts, Inc.
- cpettus@pgexperts.com
- thebuild.com
- @xof on Twitter.
What’s on the menu?

- What is a schemaless database?
- How can you use PostgreSQL to store schemaless data?
- How does do the various schemaless options perform?
A note on NoSQL.

• Worst. Term. Ever.

• It’s true that all modern schemaless databases do not use SQL, but…

• Neither did Postgres before it became PostgreSQL. (Remember QUEL?)

• The defining characteristic is the lack of a fixed schema.
A **schema** is a fixed (although mutable over time) definition of the data.

Database to schema (unfortunate term) to table to field/column/attribute.

Individual fields can be optional (NULL).

Adding new columns requires a schema change.
Rock-n-Roll!

- Schemaless databases store “documents” rather than rows.
- They have internal structure, but...
- ... that structure is per document.
- No fields! No schemas! Make up whatever you like!
We are not amused.

- Culturally, very different from the glass house data warehouse model.
- Grew out of the need for persistent object storage…
- … and impatience with the (perceived) limitations of relational databases and object-relational managers.
Let us never speak of this again.

- There’s a lot to talk about in schemaless vs traditional relational databases.
- But let’s not.
- Today’s topic: If you want to store schemaless data in PostgreSQL, how can you?
- And what can you expect?
What is schemaless data?

- Schemaless does not mean unstructured.
- Each “document” (=record/row) is a hierarchical structure of arrays and key-value pairs.
- The application knows what to expect in one of these...
- … and how to react if it doesn’t get it.
PostgreSQL has you covered.

- Not one, not two, but three different document types:
  - XML
  - hstore
  - JSON
- Let’s see what they’ve got.
It seemed like a good idea at the time.
XML

• Been around since the mid-1990s.

• Hierarchical structured data based on SGML.

• Underlying technology for SOAP and a lot of other stuff that was really popular for a while.

• Still super-popular in the Java world.
XML, your dad’s document language.

- Can specify XML schemas using DTDs.
- No one does this.
- Can do automatic transformations of XML into other markups using XSLT.
- Only the masochistic do this.
- Let’s not forget the most important use of XML!
Tomcat Configuration Files.

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Service name="Tomcat-Standalone">
    <Connector className="org.apache.catalina.connector.http.HttpConnector"
      port="8080" minProcessors="5" maxProcessors="75"
      enableLookups="true" redirectPort="8443"
      acceptCount="10" debug="0" connectionTimeout="60000"/>
    <Engine name="Standalone" defaultHost="localhost" debug="0">
      <Logger className="org.apache.catalina.logger.FileLogger"
        prefix="catalina_log." suffix=".txt"
        timestamp="true"/>
      <Realm className="org.apache.catalina.realm.MemoryRealm"/>
      <Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
        <Valve className="org.apache.catalina.valves.AccessLogValve"
          directory="logs" prefix="localhost_access_log." suffix=".txt"
          pattern="common"/>
        <Logger className="org.apache.catalina.logger.FileLogger"
          directory="logs" prefix="localhost_log." suffix=".txt"
          timestamp="true"/>
        <Context path="/examples" docBase="examples" debug="0"
          reloadable="true">
          <Logger className="org.apache.catalina.logger.FileLogger"
            prefix="localhost_examples_log." suffix=".txt"
            timestamp="true"/>
        </Context>
      </Host>
    </Engine>
  </Service>
</Server>
```
XML Support in PostgreSQL.

- Built-in type.
- Can handle documents up to 2 gigabytes.
- A healthy selection of XML operators.
  - xpath in particular.
- Very convenient XML export functions.
- Great for external XML requirements.
XML Indexing.

- There isn’t any.
- Unless you build it yourself with an expression index.
- Functionality is great.
- Performance is... we’ll talk about this later.
hstore

The hidden gem of contrib/
**hstore**

- A key/value storage type specific to PostgreSQL.
- Maps string keys to string values.
- Contrib module; not part of the PostgreSQL core.
hstore functions

• Lots and lots and lots of hstore functions.
• h->"a" (get value for key a).
• h?"a" (does h contain key a?).
• h@>"a->2" (does key a contain 2?).
• Many others.
hstore indexing.

• Can create GiST and GIN indexes over hstore values.

• Indexes the whole hierarchy, not just one key.

• Accelerates @>, ?, ?& and ?| operators.

• Can also build expression indexes.
JSON

All the *cool* kids are doing it.
JSON

- JavaScript Object Notation.
- JavaScript’s data structure declaration format, turned into a protocol.
- Dictionaries, arrays, primitive types.
- Originally designed to just be passed into `eval()` in JavaScript.
- Please don’t do this.
JSON, the new hotness

- The de facto standard API data format for REST web services.
- Very comfortable for Python and Ruby programmers.
- MongoDB's native data storage type.
JSON? Yeah, we got that.

- JSON type in core as of 9.2.
- Validates JSON going in.
- And... not much else right now.
  - array_to_json, row_to_json.
- Lots more coming in 9.3 (offer subject to committer approval).
JSON Indexing.

- Expression indexing.
- Can also treat as a text string for strict comparison...
  - … which is kind of a weird idea and I’m not sure why you’d do that.
- But the coolest part of JSON in core is!
PL/V8!

• The V8 JavaScript engine from Google is available as an embedded language.

• JavaScript deals with JSON very well, as you’d expect.

• Not part of core or contrib; needs to be built and installed separately.
PL/V8 ProTips

- Use the static V8 engine that comes with PL/V8.
- Function is compiled by V8 on first use.
- Now that we got rid of SQL injection attacks, we now have JSON injection attacks.
- PL invocation overhead is non-trivial.
Schemaless Strategies

• Create single-field tables with only a hierarchical type.
• Wrap up the (very simple) SQL to provide an object API.
• Create indexes to taste
• Maybe extract fields if you need to JOIN.
• Profit!
CREATE OR REPLACE FUNCTION get_json_key(structure JSON, key TEXT) RETURNS TEXT AS $get_json_key$
    var js_object = structure;
    if (typeof ej != 'object')
        return NULL;
    return JSON.stringify(js_object[key]);
$get_json_key$
    IMMUTABLE STRICT LANGUAGE plv8;
CREATE TABLE blog {
  post json
}

CREATE INDEX post_pk_idx ON
  blog((get_json_key(post, 'post_id')::BIGINT));

CREATE INDEX post_date_idx ON
  blog((get_json_key(post, 'post_date')::TIMESTAMP TZ));
But but but but…

- PostgreSQL was not designed to be a schemaless database.
- Wouldn’t it be better to use a bespoke database designed for this kind of data?
- Well, let’s find out!
Some Numbers.

When all else fails, measure.
• A very basic document structure:
  • id, name, company, address1, address2, city, state, postal code.
  • address2 and company are optional (NULL in relational version).
  • id 64-bit integer, all others text.
  • 1,780,000 records, average 63 bytes each.
The Competitors!

- Traditional relational schema.
- hstore (GiST and GIN indexes).
- XML
- JSON
  - One column per table for these.
- MongoDB
Timing Harness.

- Scripts written in Python.
- psycopg2 2.4.6 for PostgreSQL interface.
- pymongo 2.4.2 for MongoDB interface.
The Test Track.

- This laptop.
- OS X 10.7.5.
- 2.8GHz Intel Core i7.
- 7200 RPM disk.
- 8GB (never comes close to using a fraction of it).
Indexing Philosophy

• For relational, index on primary key.
• For hstore, index using GiST and GIN (and none).
• For JSON and XML, expression index on primary key.
• For MongoDB, index on primary key.
• Indexes created before records loaded.
Your Methodology Sucks.

- Documents are not particularly large.
- No deep hierarchies.
- Hot cache.
- Only one index.
- No joins.
- No updates.
The Sophisticated Database Tuning Philosophy.

• None.
• Stock PostgreSQL 9.2.2, from source.
  • No changes to postgresql.conf
• Stock MongoDB 2.2, from MacPorts.
  • Fire it up, let it go.
First Test: Bulk Load

- Scripts read a CSV file, parse it into the appropriate format, INSERT it into the database.
- We measure total load time, including parsing time.
- (COPY will be much much much much much faster.)
- mongoimport too, most likely.
Observations.

- No attempt made to speed up PostgreSQL.
- Synchronous commit, checkpoint tuning, etc.
- GIN indexes are really slow to build.
- The XML xpath function is probably the culprit for its load time.
Next Test: Disk Footprint.

- Final disk footprint once data is loaded.
- For PostgreSQL, reported database sizes from the `pg_*_size` functions.
- For MongoDB, reported by `db.stats()`.
Disk Footprint in Megabytes

- Relational
- hstore
- hstore (GiST)
- hstore (GIN)
- XML
- JSON
- MongoDB

Data
Index
Observations.

- GIN indexes are really big on disk.
- PostgreSQL's relational data storage is very efficient.
- None of these records are TOAST-able.
- MongoDB certain likes its disk space.
- padding factor was 1, so it wasn’t that.
Next Test: Query on Primary Key

- For a sample of 100 documents, query a single document based on the primary key.
- Results not fetched.
- For PostgreSQL, time of `execute()` method from Python.
- For MongoDB, time of `.fetch()` method.
Fetch Time in Milliseconds (<100ms)

- Relational: 6.5 milliseconds
- XML: 6.5 milliseconds
- JSON: 9.75 milliseconds
- MongoDB: 13 milliseconds
Fetch Time in Milliseconds (>100ms)

hstore

hstore (GiST)

hstore (GIN)
Observations.

- B-tree indexes kick ass.
- GiST and GIN not even in same league for simple key retrieval.
- Difference between relational, XML and JSON is not statistically significant.
- Wait, I thought MongoDB was supposed to be super-performant. Huh.
Next Test: Query on Name

• For a sample of 100 names, query all documents with that name.
• Results not fetched.
• Required a full-table scan (except for hstore with GiST and GIN indexes).
• Same timing methodology.
Fetch Time in Milliseconds

- Relational
- hstore
- hstore (GiST)
- hstore (GIN)
- XML
- JSON
- MongoDB
Fetch Time in Milliseconds (<500ms)

- Relational
- hstore
- hstore (GiST)
- hstore (GIN)
- MongoDB
Fetch Time in Milliseconds (>500ms)

- **XML**: 50000 ms
- **JSON**: 12500 ms
Observations.

- GiST and GIN accelerate every field, not just the “primary” key.
- Wow, executing the accessor function on each XML and JSON field is slow.
- MongoDB’s grotesquely bloated disk footprint hurts it here.
- Not that there’s anything wrong with that.
Now that we know this, what do we know?
Some Conclusions.

• PostgreSQL does pretty well as a schemaless database.

• Build indexes using expressions on commonly-queried fields…

• … or use GiST and hstore if you want full flexibility.

• GIN might well be worth it for other cases.
Some Conclusions, 2.

• Avoid doing full-table scans if you need to use an accessor function.
• Although hstore’s are not bad compared to xpath or a PL.
• Seriously consider hstore if you have the flexibility.
• It’s really fast.
Flame Bait!

- MongoDB doesn’t seem to be more performant than PostgreSQL.
- And you still get all of PostgreSQL’s goodies.
- Larger documents will probably continue to favor PostgreSQL.
- As will larger tables.
You can find workloads that “prove” any data storage technology is the right answer.

dBase II included.

Be very realistic about your workload and data model, now and in the future.

Test, and test fairly with real-world data in real-world volumes.
The Future is Now!

• Many more JSON functions in PostgreSQL 9.3, now in beta.
• Better, faster support for indexing.
• No GIST or GIN indexes yet, sadly.
mongres!

- Direct Mongo wire-protocol support in PostgreSQL.
- Uses PostgreSQL’s custom background process feature.
- Prototype, but check it out.
Thank you!

thebuild.com
@xof