Joining the API billionaires’ club with Python and PostgreSQL

PyPGDay 2013
Fazal Majid, CTO, Apsalar Inc.
Who am I?

- Architect for Wanadoo, second largest ISP in Europe
- Founder and CTO of Kefta (web personalization), 2000–2007
- Founder and CTO of Apsalar since 2010
- Python user since 1993 (has it been 20 years already?)
- 18 years of Oracle experience
- Using PostgreSQL for development since then
What is Apsalar?

- Mobile analytics and targeting
- Our customers are iOS and Android app developers
- We help them better understand their users, and acquire quality new customers
- Several billion API calls per day
Our stack

- Python
- C
- nginx + HAproxy
- Redis
- Wordpress
- Zabbix
- R
- PostgreSQL
- PL/pgSQL
- PL/Proxy
- pgbouncer
- PgQ
- Londiste
- OpenIndiana on x64
Why Python & PostgreSQL?

- Everyone knows Python doesn’t scale! Something about a GIL?
- All the cool kids do Node.js/Scala/Erlang!
- But PostgreSQL is not Web scale!!
- All the cool kids use NoSQL!
- Big Data == Hadoop, right?
- The Object-relational impedance mismatch will kill you!
Why PostgreSQL?

- When scalability is the issue, you approach the DB on its own terms
- Mature & stable, does not require coders to operate
- Sharding works for our use case
- Supports transactional and analytic queries
- Free (as in beer)
- Modest ops demands—it just works
Why Solaris?

- OpenIndiana b151 (free from Oracle’s clutches)
- Robust and stable
- ZFS, specially snapshots, compression and data integrity
- Zones: lightweight virtualization
- SMF auto-restarts services and handles dependencies
- DTrace gives ultimate observability into the system

I would not necessarily recommend OpenIndiana/SmartOS/OmniOS for someone starting afresh and without prior Solaris experience.

ZFS lzjb compression effectively doubled our PostgreSQL throughput
Recipes for scalability

- Manage technical risk
- Scale your database with shared-nothing parallelism
- Asynchronous processing wherever possible
- Metrics-driven performance monitoring
- Control your dependencies stack and infrastructure
- Plan your load-shedding strategy ahead of time
Skype architecture

- Heavily based on stored procedures
- PgQ: high performance transactional message queue
- Uses PG MVCC internals for speed
- Londiste replication
- pgbouncer: multiplex funnel for database connections

PL/Proxy

- Takes care of a lot of distributed DB setup
- Sharding/routing
- Map/reduce
- You still need to watch out for deadlocks
- We route using a nginx module rather than PL/Proxy
Benefits of stored procedures

- Avoid multiple round-trips between client and DB
- reduce latency
- avoid context switches
- Better security: a SECURITY DEFINER stored procedure can perform CRUD operations on a table without having INSERT or even SELECT privileges
- Statistics using track_functions=pl and pg_stat_user_functions
Asynchronous processing

- Absorb traffic spikes
- Allow maintenance windows (just pause the queue)
- Queues with multiple workers can reduce lock contention if you can route all work that would acquire the same lock to the same worker
- Move out as much work out of synchronous calls as possible
- Unfortunately interoperable PostgreSQL MQ options are limited

OmniTI has a ZeroMQ extension for PG, unclear how stable it is
Example: launching an app

- Create a row in the sessions table
- See if certain SDK options need to be disabled
- Check if this is the first time the app was launched, if yes, check for referral source
- Update real-time cohorts
- Only the first 2 need to be done synchronously
Message queues we use

- Tailing nginx log files
- PgQ
- Redis
- LISTEN/NOTIFY
- ZeroMQ
- RabbitMQ

We considered ZeroMQ and RabbitMQ, but held off for now to avoid adding complexity and depending on software we don’t have solid operational experience with.

Tailing log files does require some attention to detail:
- keep persistent track of inode to file offset
- mutex to prevent 2 ETL jobs from running concurrently
- end processing after 5 minutes, so the script can be changed if necessary
- auto-rotate the log file when it exceeds a certain size, and process that rotated log to the bitter end
Replication: right tools for each job

- Londiste
  - Trigger-based + PgQ
  - Streaming replication
  - Whole-database only
  - Good for disaster recovery
  - Helps split OLTP and Analytic workloads

- Lazy pull-on-demand
  - When replication delay is unacceptable
  - Data-driven
  - Application-specific code required
  - Heterogeneous replication possible
  - Cacheing
Pull-on-demand

- A new session is reported to a shard
- The shard checks the ID for the app against its partial replica of the Applications table
- If not present, it asks the master DB (via PL/Proxy)
- The master DB inserts a row in its authoritative copy of the Applications table if necessary, and sends the result back to the shard
- The shard copies the row in its copy, and proceeds with the insert in the sessions table
- We still need a TRIGGER ON UPDATE on the master
Sharding with PL/Proxy

- Rare example of shard-to-shard communications

- Locates the target shard using the hash function aphash() applied to _id

- TARGET prevents a namespace collision between s_device_bind() and device_bind()

- _hopcount is incremented to detect loops

```sql
CREATE OR REPLACE FUNCTION s_new_session_simplified(
    _owner      BIGINT,
    _platform   VARCHAR(32),
    _id         VARCHAR(255),
    _keyspace   CHAR(4),
    _dkeys      TEXT[],
    _app_name   VARCHAR(255),
    _longname   VARCHAR(255),
    _version    VARCHAR(32),
    _time       TIMESTAMP WITH TIME ZONE,
    _ip         INET,
    _country    CHAR(2),
    _hopcount   INTEGER,
    OUT _shard   INTEGER,
    OUT _device BIGINT
) RETURNS RECORD AS $$
CLUSTER 'shards';
RUN ON aphash($3);
TARGET new_session_simplified;
$$ LANGUAGE plproxy SECURITY DEFINER;
```
Map-reduce with SQL

CREATE OR REPLACE FUNCTION refer_report_shard
(_owner            BIGINT,
_start            DATE,
_end            DATE,
_granularity      TEXT,
_show_campaign     BOOLEAN,
_show_placement    BOOLEAN,
_show_network      BOOLEAN,
_show_country      BOOLEAN,
_campaign          VARCHAR(255)[],
_placement          VARCHAR(255)[],
_network           VARCHAR(255)[],
_country           TEXT[],
OUT origin          DATE,
OUT campaign       VARCHAR(255),
OUT placement       VARCHAR(255),
OUT network         VARCHAR(255),
OUT country         TEXT,
OUT impressions    BIGINT,
OUT clicks          BIGINT,
OUT downloads       BIGINT,
OUT revenue         BIGINT)
RETURNS SETOF RECORD AS $$
CLUSTER 'shards';
RUN ON ALL;
$$ LANGUAGE plproxy;

CREATE OR REPLACE FUNCTION refer_report
(_owner            BIGINT,
_start            DATE,
_end            DATE,
_granularity      TEXT,
_show_campaign     BOOLEAN,
_show_placement    BOOLEAN,
_show_network      BOOLEAN,
_show_country      BOOLEAN,
_campaign          VARCHAR(255)[],
_placement          VARCHAR(255)[],
_network           VARCHAR(255)[],
_country           TEXT[],
OUT origin          DATE,
OUT campaign       VARCHAR(255),
OUT placement       VARCHAR(255),
OUT network         VARCHAR(255),
OUT country         TEXT,
OUT impressions    BIGINT,
OUT clicks          BIGINT,
OUT downloads       BIGINT,
OUT revenue         BIGINT)
RETURNS SETOF RECORD AS $$
SELECT origin, campaign, placement, network, country,
SUM(impressions)::BIGINT impressions,
SUM(clicks)::BIGINT clicks,
SUM(downloads)::BIGINT downloads,
SUM(revenue)::BIGINT revenue
FROM refer_report_shard($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
GROUP BY 1, 2, 3, 4, 5;
$$ LANGUAGE SQL SECURITY DEFINER;

CREATE OR REPLACE FUNCTION refer_report_shard
(_owner            BIGINT,
_start            DATE,
_end            DATE,
_granularity      TEXT,
_show_campaign     BOOLEAN,
_show_placement    BOOLEAN,
_show_network      BOOLEAN,
_show_country      BOOLEAN,
_campaign          VARCHAR(255)[],
_placement          VARCHAR(255)[],
_network           VARCHAR(255)[],
_country           TEXT[],
OUT origin          DATE,
OUT campaign       VARCHAR(255),
OUT placement       VARCHAR(255),
OUT network         VARCHAR(255),
OUT country         TEXT,
OUT impressions    BIGINT,
OUT clicks          BIGINT,
OUT downloads       BIGINT,
OUT revenue         BIGINT)
RETURNS SETOF RECORD AS $$
CLUSTER 'shards';
RUN ON ALL;
$$ LANGUAGE plproxy;

CREATE OR REPLACE FUNCTION refer_report
(_owner            BIGINT,
_start            DATE,
_end            DATE,
_granularity      TEXT,
_show_campaign     BOOLEAN,
_show_placement    BOOLEAN,
_show_network      BOOLEAN,
_show_country      BOOLEAN,
_campaign          VARCHAR(255)[],
_placement          VARCHAR(255)[],
_network           VARCHAR(255)[],
_country           TEXT[],
OUT origin          DATE,
OUT campaign       VARCHAR(255),
OUT placement       VARCHAR(255),
OUT network         VARCHAR(255),
OUT country         TEXT,
OUT impressions    BIGINT,
OUT clicks          BIGINT,
OUT downloads       BIGINT,
OUT revenue         BIGINT)
RETURNS SETOF RECORD AS $$
SELECT origin, campaign, placement, network, country,
SUM(impressions)::BIGINT impressions,
SUM(clicks)::BIGINT clicks,
SUM(downloads)::BIGINT downloads,
SUM(revenue)::BIGINT revenue
FROM refer_report_shard($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
GROUP BY 1, 2, 3, 4, 5;
$$ LANGUAGE SQL SECURITY DEFINER;
Python decorators

- Encapsulate common patterns:
- connection pooling
- error handling
- calling stored procedures with the same params as the Python function
- cacheing
- telemetry
- Can be stacked

```python
def pg_call(fn):
    def wrapper(*args, **kwargs):
        """Call a method or function with a master DB cursor as first argument, and other arguments following. Handles exceptions""
        with __main__.pg_sema:
            db = __main__.pg.getconn()
c = db.cursor()
        try:
            try:
                # if the server was restarted, we may need to cycle through all the connections until we get a good one
                for retry in xrange(param.pg_maxconn):
                    try:
                        return fn(c, *args, **kwargs)
                    except (psycopg2.OperationalError, psycopg2.InterfaceError) as e:
                        logging.warning('discarding pg conn due to %r' % e)
                    try:
                        __main__.pg.putconn(db, close=True)
                    except psycopg2.InterfaceError:
                        # a connection may already have been closed when it threw the exception, so we have to be prepared to this exception when we try to close it again
                        pass
                        db = __main__.pg.getconn()
c = db.cursor()
                except Exception as e:
                    raise
            except psycopg2.Error, e:
                if pg_unrecoverable(e):
                    c.execute('rollback')
                    logging.exception('Unexpected PostgreSQL exception')
                    raise
        finally:
            c.close()
            __main__.pg.putconn(db)
        wrapper.func_name = fn.func_name + '_pg'
        return wrapper
```
More decorators

- Reduce amount of boilerplate so a stored procedure can be wrapped effortlessly as a Python function

```python
def pg_proc(fn):
    """Turn an empty function into a stored procedure call""
    def wrapper(c, *args, **kwargs):
        c.callproc(fn.func_name, tuple(args))
        return c.fetchall()
    wrapper.func_name = fn.func_name + '_proc'
    return wrapper
```
Telemetry decorator

• We have a Python extension module written in C for telemetry

• Number of calls

• Mean/Max and 95th percentile response time using the Chlamtac-Jain P² algorithm

```python
def watch_time(fn):
    def wrapper(*args, **kwargs):
        before = time.time()
        try:
            return fn(*args, **kwargs)
        finally:
            delay = time.time() - before
            if delay > 5:
                logging.warning('long call %.2fms to %s %r %r'
                                 % (1000 * delay, fn.func_name, args, kwargs))
            # telemetry
            if not stats.has_key(fn.__name__):
                stats[fn.__name__] = telemetry.Telemetry()
            stats[fn.__name__].observation(delay)
        wrapper.func_name = fn.func_name + '_watch'
    return wrapper
```
Caching decorator

- Started with Memcached, quickly switched to Redis
- The hard part of caching is actually cache invalidation
- Our solution: triggers + pg_notify
- mckey() turns a function + args into a key for Redis

```python
def cached(fn):
    def wrapper(*args, **kwargs):
        # XXX we are making the assumption keyword arguments do not influence
        # XXX the result of the query
        # mckey() converts function and args into a string for use as a key
        key = mckey(fn, args)
        try:
            cached = __main__.rd.get(key)
        except redis.RedisError:
            cached = None
        if not cached:
            result = fn(*args, **kwargs)
            try:
                if result is not None:
                    __main__.rd.set(key, cPickle.dumps(result, -1))
            except redis.RedisError:
                pass
            return result
        else:
            result = cPickle.loads(cached)
            return result
        wrapper.func_name = fn.func_name + '_cached'
    return wrapper
```
Putting it all together

```sql
CREATE OR REPLACE FUNCTION get_app_store(
    _api_key TEXT,
    _platform TEXT,
    _longname TEXT
) RETURNS TEXT AS $$
DECLARE
    _result TEXT := NULL;
BEGIN
    SELECT store_url INTO _result
    FROM applications
    JOIN api_keys USING (owner)
    WHERE login=_api_key AND platform=_platform::platform AND longname=_longname
    AND version='__ALL__';
    IF FOUND THEN RETURN _result; END IF;
    -- the applications row does not exist in this shard yet, pull it
    PERFORM get_app_id(owner, _platform, 'Unknown', _longname, '__ALL__', NULL, NULL)
    FROM api_keys
    JOIN wp_users ON owner="ID"
    WHERE login=_api_key;
    SELECT store_url INTO _result
    FROM applications
    JOIN api_keys USING (owner)
    WHERE login=_api_key AND platform=_platform::platform AND longname=_longname
    AND version='__ALL__';
    RETURN _result;
END $$
LANGUAGE PLpgSQL SECURITY DEFINER;
```
Cache invalidation

CREATE OR REPLACE FUNCTION update_application(
  _id       INTEGER,
  _platform VARCHAR(32),
  _owner    BIGINT,
  _name     VARCHAR(255),
  _longname VARCHAR(255),
  _version  VARCHAR(32),
  _display  VARCHAR(255),
  _parent   INTEGER,
  _status   TEXT,
  _currency CHAR(3),
  _scheme   VARCHAR(255),
  _store_url VARCHAR(255),
  _sdk      VARCHAR(32)
) RETURNS SETOF RECORD AS $$
DECLARE
  _api_key TEXT;
BEGIN
  UPDATE applications
  SET status = _status::app_status, currency = _currency, scheme = _scheme,
  sdk = _sdk, store_url = _store_url
  WHERE id=_id;
  PERFORM pg_notify('invalidate', 'app_meta:' || _owner);
  SELECT user_login INTO _api_key
  FROM wp_users
  WHERE ID=_owner;
  PERFORM pg_notify('invalidate', 'get_app_store:' || _api_key || '_' || _platform || '_' || _longname);
$$ LANGUAGE PLpgSQL;

CREATE OR REPLACE FUNCTION applications_trigger_after() RETURNS TRIGGER AS $$
BEGIN
  -- propagate changes to the shards
  IF NEW.status IS DISTINCT FROM OLD.status
  OR NEW.scheme IS DISTINCT FROM OLD.scheme
  OR NEW.store_url IS DISTINCT FROM OLD.store_url
  OR NEW.sdk IS DISTINCT FROM OLD.sdk
  OR NEW.currency IS DISTINCT FROM OLD.currency
  THEN
    PERFORM * FROM update_application(NEW.id, NEW.platform::TEXT, NEW.owner, NEW.name, NEW.longname, NEW.version, NEW.display, NEW.parent, NEW.status::TEXT, NEW.currency, NEW.scheme, NEW.store_url, NEW.sdk)
    AS sop (INT);  
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE PLpgSQL;

CREATE OR REPLACE FUNCTION update_application(
  _id       INTEGER,
  _platform VARCHAR(32),
  _owner    BIGINT,
  _name     VARCHAR(255),
  _longname VARCHAR(255),
  _version  VARCHAR(32),
  _display  VARCHAR(255),
  _parent   INTEGER,
  _status   TEXT,
  _currency CHAR(3),
  _scheme   VARCHAR(255),
  _store_url VARCHAR(255),
  _sdk      VARCHAR(32)
) RETURNS SETOF RECORD AS $$
CLUSTER 'shards';
RUN ON ALL;
$$ LANGUAGE plproxy;
#!/usr/bin/env python
#
# invalidate daemon - listen to PostgreSQL NOTIFY and delete the corresponding
# Redis key
#
import sys, os, logging, logging.config, getopt, redis, psycopg2, select
import param

if __name__ == '__main__':
    rd = redis.Redis(db=param.redis_db)
    db = psycopg2.connect(param.db)
    db.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    c = db.cursor()
    c.execute('listen invalidate')
    # Requires psycopg2 2.4
    # see: http://initd.org/psycopg/docs/advanced.html#asynchronous-notifications
    while True:
        if select.select([db], [], [], 5) == ([], [], []):
            pass
        else:
            db.poll()
        while db.notifies:
            event = db.notifies.pop()
            logging.info('invalidate %r' % event.payload)
            rd.delete(event.payload)

Invalidated daemon

Listens for notifications on the channel ‘invalidate’ and deletes those keys from Redis
Bonus feature: AWS blues

- AWS/EC2 is the roach motel of cloud services
- Inconsistent performance across instances or across time
- Mostly poor disk and network I/O performance
- Weird and frequent failure modes
- More expensive than colo well before > $10K/month
- Migrating non-trivial amounts of data is really hard
If you must use AWS

- Avoid EBS like the plague, use instance storage instead
- Use streaming replication across availability zones for disaster recovery
- Use modern instance types less likely to be hosted on tired hardware
- Test your instances and reject runts (Gresham’s Law)
  [http://majid.info/blog/aws-gresham/](http://majid.info/blog/aws-gresham/)
- Start planning your exit strategy:
Futures

- Migration to all-SSD for OLTP database instances (Intel 910 PCIe)
- Currently use SATA SSD (ZFS L2ARC) and DDRrive x1 (ZFS logzilla) with SAS hard drives
- Should replace Redis for some in-memory low-latency workloads
- Use PL/V8 for in-DB JSON processing
- Use PostgreSQL-HLL (HyperLogLog algorithm) to estimate unique users
Wishlist

• Better lock contention & deadlock diagnostics
• Active/standby failover in PL/Proxy or pgbouncer
• Better PL/pgSQL debugging tools
• Native Redis interface in PostgreSQL
• Partitioning usability enhancements
• Better timeout/HOL/load-shedding in PL/Proxy
We are hiring!

- Based in San Francisco, near South Park