Hello

My name is Adam Lowry; I'm a co-founder and developer at Urban Airship in Portland, Oregon. Today I'm going to talk about our adventures with databases as our startup grew from four guys hacking as fast as they can and hustling for every customer.

A Growing Startup

We power the world's most successful mobile apps.

The Urban Airship platform is the engagement and monetization engine behind thousands of the world's most successful mobile apps.

Postgres at Urban Airship by Adam Lowry - Postgres Open, 2011-09-16

Urban Airship is a mobile services platform, providing tools for engagement and monetization for mobile application developers.

We have web services and libraries to do push notifications -- short messages delivered over the data network on behalf of an application -- and content-delivery for in-app purchases.

How did it all start?

- May 2009: Steven Osborn, Michael Richardson, Scott Kveton, and myself start Urban Airship
- June 2009: iOS 3.0 released; our system powered push notifications for the first app in the store using push: Tap Tap Revenge 2
- “Can we do this in 3 days?” “Yes.” “We have *N* million active users.” “Y- um. Yes.”

The four of us started in May of 2009
Steven and Scott had left our previous employer, but Michael and I waited it out
Steven had been doing some part time server work with an indie iOS game studio. When the iOS 3.0 developer betas were released he looked at the In-App Purchase guide and said, I don’t want to do this -- wait, other people won’t want to do this either.
- First real customer, Tapulous. “Can we do this in 3 days?” “Yes.” “We have X million active users.” “Y- um. Yes.”

Staying ahead of the avalanche

- August 2009: Sent over one million production notifications
- August 2011: Sent over five **billion** production notifications
- Tracking over 300 million installations

We were clearly in the right place at the right time.A little while ago we passed the 5 billion marker, which was pretty exciting for us. We've also released several products -- rich push, which I'll mention later, subscriptions, and we expanded our push service to Android and BlackBerry

5 billion is a great, round number, but sending notifications alone doesn't require much in the way of data storage -- what matters is the number of installations. Each of those 300 million installations is a potential recipient, and we need to have metadata on that recipient indexed and ready to decide whether a particular API call results in a message to them.

To clarify, this doesn't mean 300 million devices, but installations of an app using us on a device.

The result is a large working set with few hot points, and pretty write heavy

My goal today is to talk about our path, and hopefully someone can learn a little from our mistakes, or at least have a little bit of an laugh.

Original Architecture

- All EC2
- Python / Django application servers
- Single PostgreSQL 8.3 instance
- pg_dump backups sent to S3, and WAL shipping for warm standby

We started out with the simplest thing we could build to provide the service. A single layer of Apache/mod_wsgi python servers, with a single DB server.
To start, no ops team. Later, no DBAs; only a dev + ops working together
For reliability we did the standard warm standby with log shipping. It was my third time setting this up, and it was a pain every time. I never got comfortable with my scripts, but the shipping itself did save us on multiple EC2-caused instances.

Device tokens

- 32 Bytes
- Same for multiple apps on the same device, but different metadata
- Application ID, token, active?, created, alias, tags

```
B4860531CD04E9D294CA0E8F209179CB006B79235C7C1CCD110B364A0658BEBF
```


To provide a frame of reference for what we were working with, here's a device token.

A device token is the unique identifier for Apple's push notification system. It's 32 Bytes, and on an installation of iOS it's the same for different apps.

For us we stored in a row with the application ID, whether it's active or not (that is, can we send to it without violating Apple's rules?), and various other bits of metadata like alias, created timestamp, and a many-to-many relationship for any tags applied to the DT by the application.

A little naive

- Data model: surrogate keys led to extra joins, inefficient indexes
- EC2 woes: very hard to get decent performance in EC2; especially with EBS volumes
- Registration API: Even device tokens that would never receive a push caused regular writes
- Preparing for Rich Push

Our initial setup was working, but it was starting to creak.
Data model
EC2/EBS
 - ec2 I/O throughput was just terrible, and EBS was worse. We ended up with 4 EBS volumes RAIDed, and still we were barely keeping up.
Come back to EC2 issues later

When we were designing our Rich Push system the scaling troubles for our Push system had me nervous.
-- describe rich push --
How was I going to take the same level of service, but store every single message? A single API call could result in 10 million writes!

Enter Mongo

- Preparing for Rich Push —

  I was totally scared.

- See Michael Schurter's talk,

  Scaling with MongoDB

  http://bit.ly/schmongodb

In late 2009/early 2010 we started work on a new product, now called Rich Push. The idea was to complement push notifications, which are fire-and-forget and can be lost if the user doesn't see them before another comes, with a persistent inbox for every user of an app.
The amount of writes necessary frightened me; a single API call could result in 5 million writes.
At the same time we wanted to add more write-heavy features to the primary push notification system, but I didn't think our current PostgreSQL installation was going to handle it at our growth rate. So we went looking for alternatives.

Mongo: the Good

- Crazy-easy setup and administration, including replication
- Flexible querying
- Fast in-place updates*
- Auto-sharding coming for future expansion

Mongo is a document based store. You have a collection, which is like a table, and a document, which is like a row. The storage format is BSON, which is a binary format similar in structure to JSON, but with more data types. It uses memory-mapped files
Setup for both developers and ops is easy, asynchronous replication setup very easy
Flexible querying
Fast in place updates with an asterisk
We were working on mongo 1.2 + 1.4 primarily; auto-sharding came in 1.6 but never worked reliably for us. Mongo 2.0 just released recently.

Mongo: the Bad

- No durability (until journal added in 1.8)
- No guarantee data was written at all unless **safe=True**
- Databases and collections are created on demand

Memory mapped files only get written when one of two things happen: you call fsync or the kernel decides to write them. That means that if a server goes down you cannot trust its data at all, and you have to promote a slave. You also have to re-sync the former master and now slave fresh.
Drivers have a **safe** flag (off by default in the versions we used); if not specified the driver won't wait for a successful ack, can lose errors
Databases + collections can be created by saving a document; a typo can result in misplaced data.
Still, we knew about all of this from testing and research. We were prepared for all of these tradeoffs. Our initial uses were so promising that we moved our primary device token data to Mongo. There was more to learn, though.

Mongo: the Ugly

- Locking: global read/write lock
- Double updating
- Indexes are very inefficient in size
- Data files get fragmented, but no online compaction — “Flip/Flop for the Win”

This might make some people cringe. Mongo has a single global read/write lock for the entire server. The effect this has is that if a write ever takes a non-trivial amount of time—page fault combined with slow disk, perhaps—everything backs up. We had high lock % when disk %util was only ~30-40%
Double updating - if an update makes a document bigger than its current size it will be moved. In a multi-update that could move every document once. Customer scenario -- duplicate processing
Flip/flop oh my god.

Cassandra!

Got that familiar pain again. Time for a new magic bullet?

- Dynamo-inspired key/value store; values have multiple columns
- Each row lives on multiple nodes, determined by RF
- Highly available with node failure, configurable consistency + redundancy

Postgres at Urban Airship by Adam Lowry - Postgres Open, 2011-09-16

What's next? At this time we had the new product and the main device token store on Mongo, but we were feeling the pain. New product comes along; with our growth we have to expect even faster growth. We need Scalability.
Cassandra came out of Facebook, but now developed as an Apache project.

Cassandra: the good

- When a single node went totally down there was no disruption
- Adding new nodes is straightforward
- Active community, commercial support

If a node goes down, no problem
Adding a new one is straightforward, although rebalancing the ring can be extremely expensive
Riptano

Cassandra: the Bad

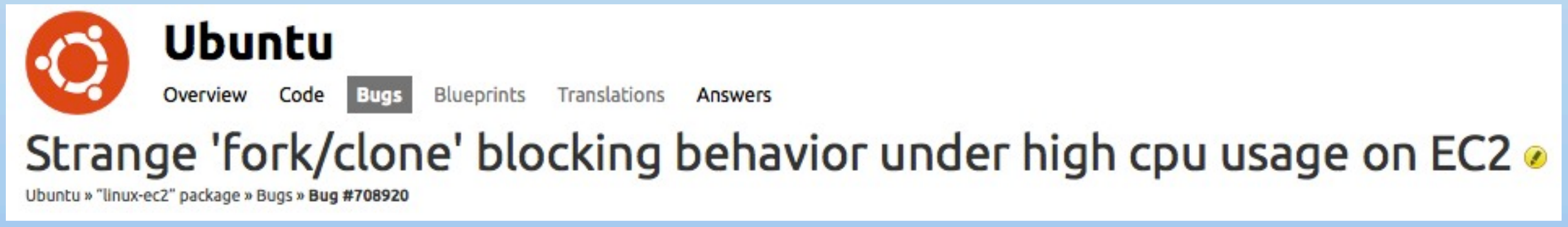
- Byzantine data model
- Manual indexing
- Counters

Cassandra: the Ugly

- Extremely young project
  - API changes
  - Buggy drivers
- Partial failures hard to route around
- Cascading failures
- Thrift

Partial failures - when EBS fails it just gets reaaally slow
Cascading failures - hinted handoff resulted in waves of stop-the-world garbage collections throughout the ring

Cassandra: the Ugly



> Based on the latest comments (unfortunately this bug was not referenced when committing the change), I am closing this bug as fixed. If anybody thinks it is still a problem, feel free to re-open it (or a new report). Thanks.

Postgres at Urban Airship by Adam Lowry - Postgres Open, 2011-09-16

Six months to fix a critical lock issue in Ubuntu's version of the Xen-ready kernel.

HBase?

HBase?

Revisiting our data

- Back to what we trusted, but taking all the knowledge we've gained.
- Plus, PostgreSQL 9.0!

After all of these paths it was time to revisit our original large dataset.
- we wanted it out of Mongo
- Couldn't really trust Cassandra, despite our admiration for it
- Back to what we trust — Postgres.

Data model/app layer changes

- BYTEAs to store the device tokens themselves
- Composite primary and foreign keys to reduce indexes
- Partial indexes to reduce size
- Custom application types for device tokens
- Manual sharding on application

SQLAlchemy / Django

- Wrote shared data acces layer using SQLAlchemy instead of Django

```
class HexBYTEA(sa.types.TypeDecorator):
    """Wrapper for byte arrays that hexify on load and dehexify on store."""

    impl = postgresql.BYTEA

    def process_bind_param(self, value, dialect):
        """Turn a hex string into bytes in before storing it in the DB."""
        return value.decode('hex')

    def process_result_value(self, value, dialect):
        """Return a hexified string to Python.

        We upper-case here to maintain our device token standards; if this gets
        used for PINs we have been keeping those as lower-case, so this might
        need tweaking.

        """
        if value is not None:
            return value.encode('hex').upper()
        else:
            return None
```


I love SQLAlchemy. Reasons we used it instead of Django's layer for this portion:

* connection pooling (pgbouncer is great, but no reason not to keep sockets open locally)
* composite keys (one of the few things you just can't do in Django)

Partial Indexes

```
CREATE INDEX device_tokens_alias
  ON device_tokens (app, alias) WHERE alias IS NOT NULL
```


Triggers for counters

```
CREATE OR REPLACE FUNCTION update_dt_stats_for_insert() RETURNS trigger AS $$
DECLARE
    counter integer;
BEGIN
    UPDATE device_token_stats
    SET
        device_tokens = device_tokens + 1,
        active_device_tokens = active_device_tokens + 1
    WHERE app = NEW.app;
    GET DIAGNOSTICS counter = ROW_COUNT;
    IF counter = 0 THEN
        INSERT INTO device_token_stats (app, device_tokens,
active_device_tokens) values (NEW.app, 1, 1);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_dt_stats_for_insert
    AFTER INSERT ON device_tokens
    FOR EACH ROW
    WHEN (NEW.active = 't')
    EXECUTE PROCEDURE update_dt_stats_for_insert();
```


Streaming queries

```
dts = session.execute(
      sa.select(
          fields,
          sa.and_(*filters)
      ).execution_options(stream_results=True)
)
```

Insert/ignore

```
CREATE OR REPLACE FUNCTION insert_ignore_tag(app VARCHAR, tag VARCHAR)
RETURNS BOOL AS
    $$
    BEGIN
        BEGIN
            INSERT INTO tags (app, tag) VALUES (app, tag);
            return true;
        EXCEPTION WHEN unique_violation THEN
            -- ignore dupes
            return false;
        END;
    END;
    $$
    LANGUAGE plpgsql;
```


Since the vast majority of activity is API-based, we have people doing very weird things that are legal but strange. An example: sending two device token registration requests at the same time with two different values.

Need to handle all sorts of concurrency errors, so all three main types we work with have insert/ignore sections.

Replication

- 9.0's streaming replication, hot standby
- Strange issues with open transactions causing replay to stop

Results?

- 120 GB of MongoDB data became 70 GB in PostgreSQL (now much larger)
- Transition took a very long time
- Performance still not good enough!

Digression: EC2

- EC2 hates your databases.
- http://bit.ly/gavin-cloud
- EBS and Ephemeral disks go over the network; network activity competes with disk.
- Use ephemeral drives, stick with XL instances (15 GB RAM, 4 disks to be RAIDed)

The Magic Bullet

What is the final magic bullet? Real hardware. I know, I know -- staggering for all of you. But even with aggressive distribution you're still hurting every time you hit the disk in EC2.
We moved to a hybrid approach with the majority of our system in a datacenter next to AWS, still have several systems in EC2.
I miss EC2's ease of use.

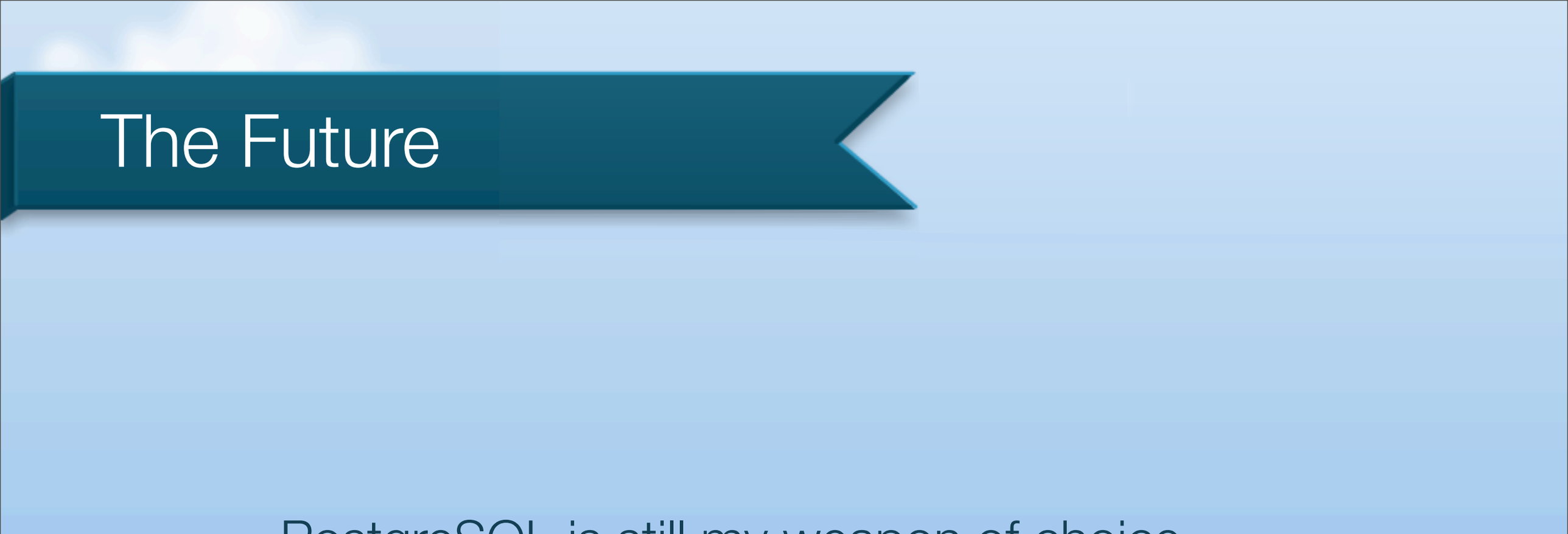
The Future

- Still need reliable, scalable data storage
- Current PostgreSQL setup is buying us some time;

  but too many manual pieces

We need more automation; we need more automatic failover, and automatic distribution. Moving apps between shards is failure prone and time consuming.

The Future



Whatever we come up with will be based on PostgreSQL; it doesn’t keep wake up my team.

Having to worry about the database distracts me from doing what I love the most about my job; I get to help other developers build their products.

Thanks!

- adam@therobots.org / adam@urbanairship.com
- http://twitter.com/robotadam
- http://urbanairship.com
- http://urbanairship.com/jobs