Migrating a live Postgres database into RDS with no downtime

Experiences and Lessons Learned

David Benoit
GoInterpay

How we thought we were going to migrate to RDS with no downtime

What actually happened when we tried it

Our Installation

- Isolated production environment in AWS
- Multiple databases
  - Live Transactions
  - Payment Details
  - FX Quotes and Trades
  - Fraud Tracking
  - Audit Records for PCI, AML, etc.
- Dedicated hosts for PostgreSQL Install
- Backups to S3

GO interpay

“No downtime”

- No disruption of the service
- 99.999% availability
  - We have limited opportunity for whole-service outages to perform upgrades or migrations
  - Evolution of the service has to be planned
- Customer Service doesn’t count
- Administrative functions don’t count
  - Fraud screening
  - Merchant access

GO interpay

payments matter

Outline

- Determine a reasonable plan
  - Migrate to Multi-AZ RDS installation
  - Change everything else after
- Set a deadline
  - It needs to happen
  - Current administration overhead is too high
- Submit a talk
  - If it has to be done, and you have a deadline, what could go wrong?

The Plan

- Create a replica of current installation
- Work out all the details
- Keep good records
  - For the talk, of course
  - and the audits
- Re-run the tests
  - To make sure it is 1am proof
  - To measure and reduce side-effects
  - And document all the steps to be 1am proof

Start with the WORM

- Dedicated database that stores events
  - All requests and responses, in and out, with detailed timing
  - No updates
  - For auditing and diagnostics
- Can afford to have delayed read updates
  - Audits and diagnostics can wait - usually
- Can NOT afford to lose writes
  - We need it all recorded

WORM Plan

- Use DMS to migrate all of the data
- Switch all reads to the replicant - the soon-to-be master
- Verify data integrity, etc.
- Switch writes
  - Bump the sequence numbers on the new master
  - Switch DNS records
  - Wait for the old database to drain
  - Let DMS finish migration

First lessons

- `max_replication_slots`
- `max_wal_senders`
  - Needs to be increased to accommodate DMS
  - Each task needs a slot
- `wal_sender_timeout`
- `hba.conf`
  - Needs to allow access from DMS instance
  - `host replication my_super_user 10.0.2.232/32 md5`
  - `my_super_user` needs replication permission

“that might be an issue” - Tim

```
2016-03-21T21:10:58 [SOURCE_UNLOAD ]W: Value for column
'Data' was truncated. data len: 252218, bind len: 65538
(ar_odbc_stmt.c:2752)
```

```
2016-03-21T21:39:07 [TARGET_LOAD ]E: Command failed to load
data with exit error code 1, Command output: ERROR: insert
or update on table "Milestones" violates foreign key
constraint "Milestones_RecordId_fkey"
```

First attempts

- First test failed
  - "`text`" is considered a CLOB type in DMS
  - Don't load your entire schema
- Second "Full LOB" test was slow
  - We let it finish, and it took 9d 20h 48m
- Third test seemed to work
  - We ran with LOB truncation set beyond largest
  - Finished in 2h 8m
- Did it work?

Reality …

- We thought it worked
- Our checks seemed to indicate it did
- We switched over the readers to use it
- Writes remained on the old master
- DMS continued to migrate new records
- Unfortunately, it corrupted some of the new records
    - We checked, and it only started after the initial load

Other things we learned

- Functions are not migrated
  - This may be problematic for you
- Indexes are not migrated
  - This is likely good, but you also need to know that you need to re-create them
- Constraints are not migrated
  - Likely to facilitate bulk data loading, but could be done after that
- Just the basic table layout

What we did

- Use `pg_dump` to get all the pieces
  - `pg_dump -s my_database > file.psql`
- Edit heavily
  - Remove table creation
  - Remove sequence updates
- Use DMS “Full Load with ongoing changes”
  - Will import all the data from when you start, they start migrating changes as they happen
- When the full load has completed, load the file

What that gets you

- Your data is loaded and changes are migrating
- Your functions are in place
- Indexes are re-created
- Constraints are back

- Basically, you have a (mostly) functional database
  - Except for the sequences

aside

- Creating indexes takes a while
- Adjust console timeouts accordingly
- Some kind of ASCII progress meter would have saved our first run

- PCI is fun!

Sequence update

something like this:

```
select max("MilestoneId") + 10000 into _seq
from timeline."Milestones";
select 'alter sequence timeline."
Milestones_MilestoneId_seq" restart with ' ||
_seq::text;
```

One last test

- This time with the right instance type
- Initial load took 29 minutes
  - We suspect the IOPS for the destination made the difference
  - db.m3.xlarge Multi-AZ vs. db.r3.2xlarge Multi-AZ
- Still started to corrupt data after the initial load
- Still didn't want to run long term without full LOB
  - We know the length of the longest existing record
  - We don't know anything about any new records

corruption

```
Data          | {"status":200,"entity":"var ...
bit_length    | 5536

Data          | '{"status":200,"entity":"var ...
bit_length    | 5544
```

Next … the important DB

- About 75 inter-related tables
  - Live transactions
  - Order details
  - Payment details
  - Fraud
  - FX quotes
  - Remittance data
  - etc.

The normal “evolved mess”

interpay

Minor detail - C

```
create or replace function uuid.generate()
returns uuid
as '$libdir/uuid-ossp', 'uuid_generate_v4'
volatile strict language C;
```

```
create or replace function uuid.generate()
returns uuid
security definer
language plpgsql
as $$
declare
begin
    return uuid_generate_v4();
end;
$$;
```

```
create or replace function uuid.generate()
returns uuid
security definer
language plpgsql
as $$
declare
begin
    return pgcrypto.gen_random_uuid();
end;
$$;
```

“that’s a killer” - Tim

```
Hstore is not a supported data type for
postgres using AWS DMS. Please find the list
of supported data types at http://docs.aws.
amazon.
com/dms/latest/userguide/CHAP_Reference.
Source.PostgreSQL.DataTypes.html .'
```

oops

```
[11:18] benoit: pg_xlog caused a drive on db1
        to go to 92% in 12 hours from 88%
```

- While you are figuring out all this stuff...

We didn’t get far

- We use some PG specific types
  - Like **HSTORE**
  - In about 5 different tables
  - DMS doesn’t like that .. yet
- We don’t have any more clever schemes
- So we stopped

Recommendations for the RDS Team

- Support all PostgreSQL data types
  - And if you can't do all of them, at least scan the schema at the start and stop
- Fix "Full LOB" migration
  - It shouldn't take 120x longer than truncating
  - Especially if 99.9% of the data is shorter than the chunk size
- DMS Instance Types
  - What is the difference?
  - No indication anywhere of what is impacted by the selection

More Recommendations

- DMS Instance storage
  - What is it for? How do I chose?
- Figure out the data corruption
  - We have no idea why it would happen
  - Nothing special about what we're doing
- Design for novice users
  - A key motivator for us was offloading the low-level details
- DMS instance couldn't resolve `ip-10-0-128-10.ec2.internal`

One more

- Fix the DMS status bar
  - Currently indicates % of tables migrated
  - In our case, sat at 0% for a while
  - Then 33% for a while
  - Then 66% for a LONG time
- Some better method?

So?

- Are we done yet?
  - No
- Can we use DMS?
  - No … not yet
- Can we use RDS?
  - Yes!
  - But we want NOTIFY/LISTEN to work soon!
- What now?
  - Old-school methods

We still have to do it

- WORM data
  - Manual replication
  - Bump the sequences
  - Update DNS
  - Backfill the updates
- Payment Database
  - Backfill as much as possible
  - Stop everything
  - dump/restore
  - Eat into our uptime budget

¯_(ツ)_/¯

better ideas?