Top 10 Migration Mistakes

From Oracle to PostgreSQL

Jim Mlodgenski, Principal Database Engineer Amazon RDS

February, 2020

Amazon Relational Database Service (Amazon RDS)

Managed relational database service with your choice of database engine

Amazon Aurora | MySQL | PostgreSQL | MariaDB | Microsoft SQL Server | Oracle

Easy to administer

Easily deploy and maintain hardware, OS and DB software; built-in monitoring

Available & durable

Automatic multi-AZ data replication; automated backup, snapshots, failover

Performant & scalable

Scale compute and storage with a few clicks; minimal downtime for your application

Secure & compliant

Data encryption at rest and in transit; industry compliance and assurance programs

aws

Why?

Project deadline

- Looming Oracle renewal

Lack of education

Attitude

- Only see the world through an Oracle lens

Using migration tools or other short cuts

aws

System Tuning

When moving to PostgreSQL, many admins start with configuring values similar to the Oracle settings

“My SGA was set to 16GB so shared_buffers is 16GB”

- Oracle’s SGA contains the Redo log buffer
- PostgreSQL has a separate WAL buffer

Tip: Use Oracle’s settings as guidance of where to start, not as a hard rule

aws

Table Spaces

- In Oracle, table spaces are critical for storing data
- Generally many table spaces are used for indexes and tables

```
CREATE TABLESPACE ts_data1
  LOGGING
  DATAFILE '/data/ts_data1.dbf'
  SIZE 32m
  AUTOEXTEND ON
  NEXT 32m MAXSIZE 2048m
  EXTENT MANAGEMENT local;
```

Table Spaces

- In PostgreSQL, table spaces are just directory locations
- Provides no real benefit unless the database spans multiple mount points

```
CREATE TABLESPACE ts_data1
  LOCATION '/data/ts_data1';
```


aws

Case Folding

In Oracle, all meta-data folds to uppercase

```
SQL> DESC PEOPLE
Name        Null?    Type
----------- -------- ------------------------
FNAME                VARCHAR2(100)
MNAME                VARCHAR2(100)
LNAME                VARCHAR2(100)
```


aws

Case Folding

In PostgreSQL, all meta-data folds to lowercase

```
test=# \d people
Table ”public.people”
 Column | Type                        | Nullable
--------+-----------------------------+----------
 fname  | character varying(100) |
 mname  | character varying(100) |
 lname  | character varying(100) |
```


aws

Case Folding

Some migration tools carry the uppercase from Oracle over to PostgreSQL

```
test=# \d ”PEOPLE”
Table ”public.PEOPLE”
 Column | Type                    | Nullable
--------+-------------------------+----------
  FNAME | character varying(100) |
  MNAME | character varying(100) |
  LNAME | character varying(100) |
```


aws

Case Folding

Becomes very tedious needing to double quote everything

```
test=# SELECT ”FNAME”, ”MNAME”, ”LNAME” FROM ”USERS”;
    FNAME  | MNAME   | LNAME
 ---------+---------+------------
 Marilyn  |         | Monroe
 Nelson   |         | Mandela
 John     | F.      | Kennedy
 Martin   | Luther  | King
 Winston  |         | Churchill
 Bill     |         | Gates
 Mahatma  |         | Gandhi
 Margaret |         | Thatcher
 Elvis    |         | Presley
 Albert   |         | Einstein
(10 rows)
```


aws

DUAL Table

In Oracle, the DUAL table is used to run functions

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
02-FEB-20
```


aws

DUAL Table

Mocking a DUAL table

```
test=> CREATE TABLE dual (dummy varchar);
CREATE TABLE

test=> INSERT into dual VALUES ('X');
INSERT 0 1

test=> SELECT current_date FROM dual;
 current_date
--------------
 2020-02-12
(1 row)
```


aws

DUAL Table

Mocking a DUAL table

```
test=> INSERT into dual VALUES ('X');
INSERT 0 1

test=> SELECT current_date FROM dual;
 current_date
--------------
 2020-02-12
 2020-02-12
(2 rows)
```


aws

DUAL Table

In PostgreSQL, the FROM clause is optional and is unnecessary

Do not mock a DUAL table

```
test=# SELECT CURRENT_DATE;

 current_date
--------------
 2020-02-12
(1 row)
```


aws

DUAL Table

If application code makes the DUAL table necessary, create it as a view instead of a table

```
test=> CREATE VIEW dual AS SELECT 'X'::varchar AS dummy;
CREATE VIEW

test=> SELECT current_date FROM dual;
 current_date
--------------
 2020-02-12
(1 row)
```


aws

Synonyms

"PostgreSQL doesn't have synonyms so I can't migrate my application"

```
CREATE PUBLIC SYNONYM emp
FOR SCOTT.emp;
```

Synonyms are used to not fully qualify cross schema objects

- Mostly a convenience feature

aws

Synonyms

In PostgreSQL, search_path can accomplish many of the same things and is less tedious to setup

```
test=# show search_path;

  search_path
-----------------
 "$user", public
(1 row)
```


aws

Synonyms

```
CREATE FUNCTION user1.get_int()
   RETURNS int AS
 $$
   SELECT 1;
 $$ LANGUAGE sql;

 CREATE FUNCTION user2.get_int()
   RETURNS int AS
 $$
   SELECT 2;
 $$ LANGUAGE sql;

 CREATE FUNCTION public.get_number()
   RETURNS float8 AS
 $$
   SELECT 3.14::float8;
 $$ LANGUAGE sql;
```

Synonyms

```
test=# SELECT get_int();
2017-05-08 17:38 EDT [28855] ERROR: function get_int() does not ...
2017-05-08 17:38 EDT [28855] HINT: No function matches the given...
2017-05-08 17:38 EDT [28855] STATEMENT: SELECT get_int();
ERROR: function get_int() does not exist
LINE 1: SELECT get_int();
               ^
HINT: No function matches the given name and argument types. You...
```

Synonyms

```
test=# SET search_path = user1, user2, public;
 SET

 test=# SELECT get_int();
  get_int
 ---------
        1
 (1 row)
```


aws

Synonyms

```
test=# SET search_path = user2, user1, public;
 SET

 test=# SELECT get_int();
  get_int
 ---------
        2
 (1 row)
```


aws

Synonyms

```
test=# select get_number();
  get_number
 ------------
        3.14
 (1 row)
```


aws

Nulls

PostgreSQL and Oracle handle nulls a bit differently

- Need to account for them appropriately
- Most often seen with string concatenation

Oracle

NULL = ‘’

PostgreSQL

NULL != ‘’

aws

Nulls

```
CREATE TABLE people (
     fname VARCHAR2(100),
     mname VARCHAR2(100),
     lname VARCHAR2(100)
 );

 SELECT fname || ' ' || mname || ' ' || lname
 FROM people;
```


aws

Nulls

```
SQL> SELECT fname || ' ' || mname || ' ' || lname FROM people;

 FNAME||''||MNAME||''||LNAME
 ---------------------------------------------------------------
 Marilyn  Monroe
 Nelson  Mandela
 John F. Kennedy
 Martin Luther King
 Winston  Churchill
 Bill  Gates
 Mahatma  Gandhi
 Margaret  Thatcher
 Elvis  Presley
 Albert  Einstein

 10 rows selected.
```


aws

Nulls

```
test=# SELECT fname || ' ' || mname || ' ' || lname FROM people;
        ?column?
 -------------------------


  John F. Kennedy
  Martin Luther King









 (10 rows)
```


aws

Nulls

```
test=# SELECT COALESCE(fname, '') || ' ' ||
        COALESCE(mname, '') || ' ' || COALESCE(lname, '') FROM people;
      ?column?
 --------------------
  Marilyn  Monroe
  Nelson  Mandela
  John F. Kennedy
  Martin Luther King
  Winston  Churchill
  Bill  Gates
  Mahatma  Gandhi
  Margaret  Thatcher
  Elvis  Presley
  Albert  Einstein
 (10 rows)
```


aws

Nulls

Tip: Built-in functions handle nulls

```
test=# SELECT concat_ws(' ', fname, mname, lname) FROM people;
      concat_ws
 --------------------
  Marilyn Monroe
  Nelson Mandela
  John F. Kennedy
  Martin Luther King
  Winston Churchill
  Bill Gates
  Mahatma Gandhi
  Margaret Thatcher
  Elvis Presley
  Albert Einstein
 (10 rows)
```

Nulls - Unique constraints

```
SQL> CREATE UNIQUE INDEX null_test_idx ON null_test (c1, c2);

Index created.

SQL> INSERT INTO null_test (c1,c2) VALUES (1, 'a');

1 row created.

SQL> INSERT INTO null_test (c1) VALUES (1);

1 row created.

SQL> INSERT INTO null_test (c1) VALUES (1);
INSERT INTO null_test (c1) VALUES (1)
*
ERROR at line 1:
ORA-00001: unique constraint (MASTER.NULL_TEST_IDX) violated
```


aws

Nulls - Unique constraints

```
=> CREATE UNIQUE INDEX null_test_idx ON null_test (c1, c2);
CREATE INDEX

=> INSERT INTO null_test (c1,c2) VALUES (1, 'a');
INSERT 0 1

=> INSERT INTO null_test (c1) VALUES (1);
INSERT 0 1

=> INSERT INTO null_test (c1) VALUES (1);
INSERT 0 1
```

Fine Tuning Queries

"There is an index hint but PostgreSQL does not use it"

PostgreSQL does not have hints as part of the core database

- It is available as an extension (pg_hint_plan)

It treats Oracle hints as comments

PostgreSQL's optimizer is different than Oracle so queries are tuned differently

Fine Tuning Queries

"I didn't index my column in Oracle,
why would I in PostgreSQL?"

PostgreSQL has more and different types of indexes than Oracle

- B-tree
- Hash
- GIN
- Bloom
- GiST
- SP-GiST
- BRIN
- RUM

aws

Fine Tuning Queries

PostgreSQL can use indexes on LIKE queries

```
 CREATE INDEX idx_users_lname
   ON users USING gin (lname gin_trgm_ops);

 EXPLAIN SELECT * FROM users WHERE lname LIKE '%ing%';

                    QUERY PLAN
-----------------------------------------------------
  Bitmap Heap Scan on users (cost=8.00..12.02 rows=1 width=654)
    Recheck Cond: ((lname)::text ~~ '%ing%'::text)
    -> Bitmap Index Scan on idx_users_lname
         (cost=0.00..8.00 rows=1 width=0)
         Index Cond: ((lname)::text ~~ '%ing%'::text)
```


aws

Not Using Native Features

PostgreSQL is more feature rich for developers than Oracle

- Stored Procedure Languages
- Foreign Data Wrappers
- Data Types
- Spatial

Not Using Native Features

```
CREATE OR REPLACE FUNCTION has_valid_keys(doc json)
  RETURNS boolean AS
$$
  if (!doc.hasOwnProperty('fname'))
    return false;

  if (!doc.hasOwnProperty('lname'))
    return false;

  return true;
$$ LANGUAGE plv8 IMMUTABLE;

ALTER TABLE people_collection
  ADD CONSTRAINT collection_key_chk
  CHECK (has_valid_keys(doc::json));
```


aws

Not Using Native Features

```
CREATE TABLE login_history (
  user_id  bigint,
  host     inet,
  login_ts timestamptz
);

SELECT user_id, count(*)
  FROM login_history
 WHERE host << '17.0.0.0/8'::inet
   AND login_ts > now() - '7 days'::interval
 GROUP BY 1;
```


aws

Exceptions

Many Oracle procedures use exceptions as part of standard practice

Some applications have exception handling in every procedure and function

Most migration tools simply translate the code to PL/pgSQL

Exceptions

```
CREATE FUNCTION get_first_name(p_lname varchar2)
  RETURN varchar2
IS
  l_fname varchar2(100);
BEGIN
    SELECT fname
      INTO l_fname
      FROM people
     WHERE lname = p_lname;

    RETURN l_fname;
EXCEPTION
    WHEN no_data_found THEN
      l_fname := null;
    RETURN l_fname;
END get_first_name;
```

Exceptions

```
CREATE FUNCTION get_first_name(p_lname varchar) RETURNS varchar
  AS $$
DECLARE
  l_fname varchar;
BEGIN
    SELECT fname
      INTO l_fname
      FROM people
     WHERE lname = p_lname;

    RETURN l_fname;
EXCEPTION
    WHEN no_data_found THEN
      l_fname := null;
    RETURN l_fname;
END$$ LANGUAGE plpgsql;
```


aws

Exceptions

PostgreSQL uses sub transactions to handle exceptions

```
SAVEPOINT hidden_savepoint;

SELECT fname
  INTO l_fname
  FROM people
 WHERE lname = p_lname;

 if exception
     ROLLBACK TO SAVEPOINT hidden_savepoint;
     l_fname := null;

 otherwise
     RELEASE SAVEPOINT hidden_savepoint;
```

Exceptions

Most exception blocks are not necessary

```
CREATE OR REPLACE FUNCTION get_first_name(p_lname varchar)
  RETURNS varchar
AS $$
DECLARE
  l_fname varchar := null;
BEGIN
    SELECT fname
      INTO l_fname
      FROM people
     WHERE lname = p_lname;

     RETURN l_fname;
END
$$ LANGUAGE plpgsql;
```


aws

Exceptions

Most exception blocks are not necessary

```
test=> SELECT get_first_name('Gates');
 get_first_name
----------------
 Bill
(1 row)

test=> SELECT get_first_name('gates');
 get_first_name
----------------

(1 row)
```


aws

Exceptions

Not all Oracle exceptions are PostgreSQL exceptions

```
CREATE FUNCTION get_first_name(p_lname varchar) RETURNS varchar
  AS $$
DECLARE
  l_fname varchar;
BEGIN
    SELECT fname
      INTO l_fname
      FROM people
     WHERE lname = p_lname;

    RETURN l_fname;
EXCEPTION
    WHEN no_data_found THEN
      l_fname := 'NOT_FOUND';
    RETURN l_fname;
END$$ LANGUAGE plpgsql;
```


aws

Exceptions

Not found and too many rows are not PL/pgSQL exceptions

```
test=> SELECT get_first_name('gates');
 get_first_name
----------------

(1 row)
```


aws

Exceptions

Use STRICT to get Oracle like behavior

```
CREATE FUNCTION get_first_name(p_lname varchar) RETURNS varchar
  AS $$
DECLARE
  l_fname varchar;
BEGIN
    SELECT fname
      INTO STRICT l_fname
      FROM people
     WHERE lname = p_lname;

     RETURN l_fname;
EXCEPTION
    WHEN no_data_found THEN
      l_fname := 'NOT_FOUND';
    RETURN l_fname;
END$$ LANGUAGE plpgsql;
```


aws

Exceptions

Use STRICT to get Oracle like behavior

```
test=> SELECT get_first_name('gates');
 get_first_name
----------------
 NOT_FOUND
(1 row)
```


aws

Data Types

Oracle has a few main data types that are typically used

- VARCHAR2
- DATE
- NUMBER

And a couple Large Object types

- CLOB
- BLOB

Data Types

PostgreSQL has 64 base types and can be extended for more

```
abstime
aclitem
Bit
Bool
Box
Bpchar
Bytea
Char
cid
Cidr
Circle
Date
Float4
Float8
Gtsvector
inet
Int2
Int2vector
Int4
Int8
Interval
Json
Jsonb
Line
Lseg
Macaddr
Money
Name
Numeric
Oid
Oidvector
path
pg_lsn
pg_node_tree
Point
Polygon
Refcursor
Regclass
Regconfig
Regdictionary
Regnamespace
Regoper
Regoperator
Regproc
Regprocedure
Regrole
Regtype
reltime
Smgr
Text
Tid
Time
Timestamp
Timestamptz
Timetz
Tinterval
Tsquery
Tsvector
txid_snapshot
Uuid
Varbit
Varchar
Xid
xml
```


aws

Data Types

The perceived equivalent in PostgreSQL does not always behave the same as Oracle

For example, managing CLOBS

- Length
- Substrings

```
DBMS_LOB.GETLENGTH(x)
```


aws

Data Types

In PostgreSQL, VARCHAR and TEXT are equivalent and behave the same

```
CREATE TABLE max_varchar (
    a  varchar(4001)
);

CREATE TABLE max_varchar (
    a  varchar(10485760)
);
```


aws

Data Types

```
test=# INSERT INTO max_varchar SELECT repeat('x', 1073741800);
INSERT 0 1

test=# SELECT length(a) from max_varchar ;
  length
------------
 1073741800
(1 row)
```


aws

Number(38)

99999999999999999999999999999999999999

Numeric

Data Types

Most migration tools translate an Oracle NUMBER to a PostgreSQL NUMERIC

A PostgreSQL NUMERIC can hold

- 131072 before the decimal point
- 16383 after the decimal point

It is not the same are NUMBER

Data Type Performance

```
CREATE TABLE t1 (c1 numeric);
CREATE TABLE t2 (c1 numeric, c2 numeric);

DO $$
BEGIN
  INSERT INTO t1
  SELECT g
    FROM generate_series(1, 1000000) g;

  FOR i IN 1..10 LOOP
    INSERT INTO t2
    SELECT g
      FROM generate_series(1, 1000000) g;
  END LOOP;
END
$$;
```


aws

Data Type Performance

```
test=> SELECT count(*)
       FROM t1
       INNER JOIN t2
       ON (t1.c1 = t2.c1);
  count
----------
 10000000
(1 row)

Time: 2757.392 ms (00:02.757)
```


aws

Data Type Performance

```
CREATE TABLE t1 (c1 bigint);
CREATE TABLE t2 (c1 bigint, c2 bigint);

DO $$
BEGIN
  INSERT INTO t1
  SELECT g
    FROM generate_series(1, 1000000) g;

  FOR i IN 1..10 LOOP
    INSERT INTO t2
    SELECT g
      FROM generate_series(1, 1000000) g;
  END LOOP;
END
$$;
```


aws

Data Type Performance

```
test=> SELECT count(*)
       FROM t1
       INNER JOIN t2
       ON (t1.c1 = t2.c1);
  count
----------
 10000000
(1 row)

Time: 1977.685 ms (00:01.978)
```


aws

Summary

- System Tuning
- Table Spaces
- Case Folding
- DUAL Table
- Synonyms
- Nulls
- Fine Tuning
- Native Features
- Exceptions
- Data Types

Thank you

We’re hiring