Top 10 Migration Mistakes From Oracle to PostgreSQL

Jim Mlodgenski, Principal Database Engineer Amazon RDS
May, 2019
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
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”
“My redo logs are 2GB so max_wal_size is 2GB”
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';
Case Folding

In Oracle, all meta-data folds to uppercase

```sql
SQL> DESC USERS
Name       Null?   Type
---------- ------- ------------------------
FNAME      VARCHAR2(100)
MNAME      VARCHAR2(100)
LNAME      VARCHAR2(100)
```
Case Folding

In PostgreSQL, all meta-data folds to lowercase

test=# \d users
Table "public.users"

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>character varying(100)</td>
<td></td>
</tr>
<tr>
<td>mname</td>
<td>character varying(100)</td>
<td></td>
</tr>
<tr>
<td>lname</td>
<td>character varying(100)</td>
<td></td>
</tr>
</tbody>
</table>
Case Folding

Some migration tools carry the uppercase from Oracle over to PostgreSQL

```
test=# \d "USERS"
Table "public.USERS"
  Column | Type                   | Nullable
----------+------------------------+----------
     FNAME | character varying(100) |          
     MNAME | character varying(100) |          
     LNAME | character varying(100) |          
```
Case Folding

Becomes very tedious needing to double quote everything

test=# SELECT "FNAME", "MNAME", "LNAME" FROM "USERS";

<table>
<thead>
<tr>
<th>FNAME</th>
<th>MNAME</th>
<th>LNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td></td>
<td>Washington</td>
</tr>
<tr>
<td>John</td>
<td></td>
<td>Adams</td>
</tr>
<tr>
<td>Thomas</td>
<td></td>
<td>Jefferson</td>
</tr>
<tr>
<td>James</td>
<td></td>
<td>Madison</td>
</tr>
<tr>
<td>James</td>
<td></td>
<td>Monroe</td>
</tr>
<tr>
<td>Andrew</td>
<td></td>
<td>Jackson</td>
</tr>
<tr>
<td>Martin</td>
<td></td>
<td>Van Buren</td>
</tr>
<tr>
<td>John</td>
<td></td>
<td>Tyler</td>
</tr>
<tr>
<td>John</td>
<td>Quincy</td>
<td>Adams</td>
</tr>
<tr>
<td>William</td>
<td>Henry</td>
<td>Harrison</td>
</tr>
</tbody>
</table>

(10 rows)
DUAL Table

In Oracle, the DUAL table is used to run functions

```sql
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
09-MAY-17
```
DUAL Table

• In PostgreSQL, the FROM clause is optional and is unnecessary
• Do not mock a DUAL table

```
test=# SELECT CURRENT_DATE;

 current_date
-----------------
 2017-05-09
(1 row)
```
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
Synonyms

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

```sql
test=# show search_path;

search_path
-------------
"$user", public
(1 row)
```
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...
test=# SET search_path = user1, user2, public;
    SET

test=# SELECT get_int();
    get_int
    --------
      1
(1 row)
test=# SET search_path = user2, user1, public;
SET

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

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

PostgreSQL and Oracle handle nulls a bit differently

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

In Oracle

```
NULL = ""
```

In PostgreSQL

```
NULL != ""
```

And NULL != NULL
CREATE TABLE users (  
    fname VARCHAR2(100),  
    mname VARCHAR2(100),  
    lname VARCHAR2(100)  
);  
SELECT fname || ' ' || mname || ' ' || lname  
FROM users;
nulls

SQL> SELECT fname || ' ' || mname || ' ' || lname FROM users;

FNAME||''||MNAME||''||LNAME
---------------------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
James Monroe
Andrew Jackson
Martin Van Buren
John Tyler
John Quincy Adams
William Henry Harrison

10 rows selected.
nulls

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>John Quincy Adams</strong></td>
</tr>
<tr>
<td><strong>William Henry Harrison</strong></td>
</tr>
<tr>
<td>(10 rows)</td>
</tr>
</tbody>
</table>
nulls

```sql
test=# SELECT COALESCE(fname, '') || ' ' || COALESCE(mname, '') || ' ' || COALESCE(lname, '') FROM users;
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>George Washington</td>
<td>John Adams</td>
<td>Thomas Jefferson</td>
<td>James Madison</td>
</tr>
<tr>
<td>James Monroe</td>
<td>Andrew Jackson</td>
<td>Martin Van Buren</td>
<td>John Tyler</td>
</tr>
<tr>
<td>John Quincy Adams</td>
<td>William Henry Harrison</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(10 rows)
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);

* 
ERROR at line 1:
ORA-00001: unique constraint (MASTER.NULL_TEST_IDX) violated
Nulls

=> 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

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

“I added a hint to use an index 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
- GiST
- SP-GiST
- BRIN
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)
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 user_collection
  ADD CONSTRAINT collection_key_chk
  CHECK (has_valid_keys(doc::json));
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;
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
CREATE FUNCTION get_first_name(p_lname varchar2) 
    RETURN varchar2 
IS 
    l_fname varchar2(100); 
BEGIN 
    SELECT fname 
        INTO l_fname 
        FROM users 
        WHERE lname = p_lname; 

    RETURN l_fname; 
EXCEPTION 
    WHEN no_data_found THEN 
        l_fname := null; 
    RETURN l_fname; 
END get_first_name;
CREATE FUNCTION get_first_name(p_lname varchar) RETURNS varchar
    AS $$
DECLARE
    l_fname varchar;
BEGIN
    SELECT fname
    INTO l_fname
    FROM users
    WHERE lname = p_lname;

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

PostgreSQL uses sub transactions to handle exceptions

```sql
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 users
    WHERE lname = p_lname;

    RETURN l_fname;
END
$$ LANGUAGE plpgsql;
```
Exceptions
Not all Oracle exceptions are PostgreSQL exceptions

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

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

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

[gnb] > SELECT get_first_name('missing');
get_first_name
--------------
(null)
(1 row)
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 users
    WHERE lname = p_lname;

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

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

131,072 Digits
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
Summary

- System Tuning
- Table Spaces
- Case Folding
- DUAL Table
- Synonyms

- Nulls
- Fine Tuning
- Native Features
- Exceptions
- Data Types
Thank you!