Porting Oracle Applications to PostgreSQL

Peter Eisentraut, credativ GmbH / credativ Ltd.

PGCon 2008

Disclaimers

- This presentation was written by a PostgreSQL expert, not an Oracle expert.
- Both Oracle and PostgreSQL are diverse, complex, and moving targets.
- Both Oracle and PostgreSQL are (probably) Turing-complete, so almost anything is "possible", but we are looking for reasonable options.

You Will See . . .

- Porting projects are hard.
- Compatibility and compatibility layers are an illusion.
- It might be better not to do it.
- But success can be very rewarding.

Outline

Outline

Syntax

Identifiers Oracle case folds to upper case, PostgreSQL to lower case. Big trouble if you mix quoted and unquoted identifiers.

Column aliases `SELECT foo [AS] bar` — Most Oracle applications omit the `AS`, but PostgreSQL requires it. Fixed in PostgreSQL 8.4.

MINUS Change to `EXCEPT`.

SQL key words Usually not a big problem, but should be kept in mind.

"FROM dual" Easy to work around (or use orafce).

Table Definition

- The usual features are mostly the same: columns, constraints, defaults.
- Data types are more work; see below.
- No fancy features like “table of type”.

Data Types: General

- Both Oracle and PostgreSQL support plenty of SQL-conforming data types.
- But usually the nonconforming ones are in wider use.
- Thin compatibility layers can usually help, but that will make your PostgreSQL application unpretty.
- A big search-and-replace is usually in order.

Data Types: Specifics

- varchar2 → varchar or text
- clob, long → varchar or text
- nchar, nvarchar2, nclob → (varchar or text)
- number → numeric or bigint or int or smallint or double precision or real (bug potential)
- binary_float/binary_double → real/double precision
- blob, raw, long raw → bytea (additional porting required)
- date → date or timestamp

Null Values

- Infamous Oracle behavior: `NULL = ''`
- Consequently, `'' = ''` is not true
- Completely weird and inconsistent
- Usually, your data will just disappear in PostgreSQL
- `transform_null_equals` does not help here
- If your application relies on any of this, you are in trouble.

Functions: General

- Function compatibility is a bottomless pit.
- PostgreSQL (+ orafce) supports many Oracle compatibility functions.
- It's easy to write your own.
- Only the special syntax is trouble.

Functions: Compatibility

For example, the following common functions are supported by PostgreSQL as well:

- substr
- to_char
- nvl, nullif (orafce)

Functions: Specifics

Manual work required here:

- `sysdate` → `current_timestamp` or `localtimestamp`

Functions: decode

```
DECODE(expr, search, expr[, search, expr...] [, default])
```

becomes

```
CASE WHEN expr THEN search .. ELSE default END
```

Default Parameters: Overview

- PostgreSQL supports neither default values for parameters nor named parameters in function calls.
- Oracle applications make ample use of both.

Default Parameters: The Easy Case

```
CREATE FUNCTION foo (a int, b int, c int = 0) ...
```

becomes

```
CREATE FUNCTION foo (a int, b int, c int) ...

CREATE FUNCTION foo (a int, b int) ...
    AS $$ SELECT foo(a, b, 0) $$;
```

Default Parameters: The Hard Case

```
CREATE FUNCTION foo (a int, b int = 5, c int = 0) ...
```

- This is only callable with named parameters.
- PostgreSQL doesn't support this.
- You will have to change your client application.
- Your project time will double.

Default Parameters: Conclusion

- Approx. 97% of applications to be ported contain issues like this.
- Client code must be reworked.
- Adding this support in PostgreSQL would be a great feature.

Sequences: Creating

Sequences are somewhat compatible . . .

- Change `NOCACHE` to `CACHE 1` (or omit).
- `MAXVALUE 999999999999999999999999` needs to be reduced.

Don't rely on the caching behavior.

Sequences: Using

- Oracle syntax: `sequence_name.nextval`
- PostgreSQL syntax: `nextval('sequence_name')`

Search-and-replace; but direct sequence calls are rare.

Outer Joins: Overview

- PostgreSQL only supports the SQL-standard outer join syntax.
- Oracle supports it since version 9.
- Most Oracle code uses the old, Oracle-specific syntax.
- Porting is usually straightforward, but requires manual work.
- Set up test queries to catch porting mistakes.

Outer Joins: Simple Example

```
SELECT * FROM a, b WHERE a.x = b.y(+)
```

becomes

```
SELECT * FROM a LEFT JOIN b ON a.x = b.y
```

Outer Joins: Complex Example

```
SELECT ...
    FROM A, B, C
    WHERE A.A_ID (+) = B.A_ID
      AND C.C_KEY(+) = B.C_KEY
```

becomes

```
SELECT ...
    FROM A
         RIGHT JOIN
         B ON (A.A_ID = B.A_ID)
         LEFT JOIN
         C ON (C.C_KEY = B.C_KEY)
```

Outer Joins: Unclear Example

```
SELECT ...
  FROM A, B, C, D, E
 WHERE A.A_ID = B.A_ID
   AND B.B_ID = C.A_ID(+)
   AND B.B_KEY = C.B_KEY(+)
   AND C.C_ID = D.C_ID(+)
   AND B.A_ID = E.A_ID(+)
   AND B.B_KEY = E.B_KEY(+)
   AND 'CONSTANT' = C.X_ID(+)
```

What's that???

Locking

- Transaction isolation, locking, SELECT FOR UPDATE behave pretty much the same.
- Oracle also defaults to read committed.
- Usually, no one at the client has ever heard of concurrency issues, so the code is likely buggy anyway.

Indexes

- Basic syntax the same:
  `CREATE INDEX name ON table (a, b)`
- Primary keys and unique constraints are automatically indexed.
- Other features are implementation-specific.
- You will have to re-tune the entire porting result anyway.

Optimizer Hints

- Delete them
- Or keep them for future investigation
- Usually useless

Date Formatting

- `TO_CHAR` is largely compatible.
- Warning: PostgreSQL version is not very robust.
- One-argument variant provided by orafce
- `NLS_DATE_FORMAT` is replaced by locale settings.

Date Arithmetic

- Usually, date arithmetic is easier in PostgreSQL, so consider a small code rewrite.
- orafce provides compatibility functions, such as `last_day`, `add_months`.
- Oracle code often does date + int ...
  - In PostgreSQL, this may become timestamp + int.
  - This doesn't work.
  - Write a custom operator or rewrite the code.

Encodings

- Both Oracle and PostgreSQL support the same ideas.
- But everything is named differently.
- Might be a good time to review the encoding and locale choices.

NLS_* vs. LC_*

Approximate analogies:

NLS_CALENDAR	—
NLS_COMP	lc_collate = 'C'
NLS_CURRENCY	lc_monetary
NLS_DATE_FORMAT	DateStyle
NLS_DATE_LANGUAGE	lc_messages, lc_time (8.4?)
NLS_LANG, NLS_LANGUAGE	LANG, client_encoding
NLS_NCHAR	—
NLS_NUMERIC_CHARACTERS	lc_numeric
NLS_SORT	lc_collate
NLS_TERRITORY	LANG, lc_*

ROWNUM and ROWID

ROWNUM:

- Use `generate_series`, or
- Rewrite and apply `LIMIT`, or
- Just handle in the client

ROWID:

- Analogous to `ctid`
- Good code should usually not use this.
- That does not prevent some from trying.

XML

(untested!)

- xmltype → xml
- extract → xpath
- XMLELEMENT, XMLATTRIBUTES, etc. are the same.
- Most functionality is different or missing in PostgreSQL.

Triggers: Declarations

Oracle uses inline trigger actions:

```
CREATE TRIGGER foo AFTER action ON table
    AS BEGIN ... END;
```

becomes

```
CREATE OR REPLACE FUNCTION foo_tg() RETURNS TRIGGER
    LANGUAGE xxx
    AS $$ ... $$;
CREATE TRIGGER foo AFTER action ON table
    EXECUTE PROCEDURE foo_tg();
```

Note: `FOR EACH STATEMENT` is the default in Oracle and PostgreSQL.

Triggers: Column-Level Triggers

Oracle supports column-level triggers:

```
CREATE TRIGGER foo BEFORE UPDATE OF column ON table
    AS BEGIN ... END;
```

becomes

```
CREATE OR REPLACE FUNCTION foo_tg() RETURNS TRIGGER
    LANGUAGE xxx AS $$
BEGIN
  IF NEW.column IS NOT DISTINCT FROM OLD.column THEN
    RETURN NEW;
  END IF;
  ... -- normal code
END;
$$;
CREATE TRIGGER foo AFTER action ON table
    EXECUTE PROCEDURE foo_tg();
```

But this doesn't catch updates to the same value.

You will need to make a choice which behavior you need.

Things That Won't Work Directly

CONNECT BY Try contrib/tablefunc.

Materialized views Write your own wrapper.

Snapshots Write your own wrapper.

Database links Use contrib/dblink plus views.

Autonomous transactions Try dblink.

Synonyms Try views or wrapper or schema path.

Partitioning Write your own system.

Outline

orafce

`http://orafce.projects.postgresql.org/`

- Large set of Oracle compatibility functions
- “dual” table
- Debian and RPM packages available
- Invaluable

ora2pg

`http://ora2pg.projects.postgresql.org/`

- Converts Oracle schema definitions
- Extracts data from Oracle database for import into PostgreSQL
- Packages available
- Invaluable

TOra

`http://tora.sourceforge.net/`

- GUI for PostgreSQL and Oracle
- Contains exploration and debugging facilities for Oracle
- Packages available, but usually without Oracle support
- Generally a bit outdated, but good for this purpose

DBD::Oracle

`http://search.cpan.org/dist/DBD-Oracle/`

- Needed for ora2pg
- Also helpful for test scripts etc.
- Building it can be challenging
- Debian and RPM packages available

Oracle Instant Client

- Needed for DBD::Oracle and TOra
- Also contains sqlplus

download from Oracle

Oracle Database Express Edition

- Use this for testing if you have no other Oracle instance.

download from Oracle

Outline

From PL/SQL to PL/pgSQL

- Compatibility isn't that great, but it's obviously the best choice.
- The PL/pgSQL parser is DAAB.
- See also `http://www.postgresql.org/docs/current/static/plpgsql-porting.html`.

Function Creation

- `CREATE FUNCTION ... RETURN type` becomes
  `CREATE FUNCTION ... RETURNS type`
- Function body must be quoted (dollar quoting).
- Various other details are incompatible:
  - `LANGUAGE`
  - `STRICT`, `STABLE`, etc.
- For variable declarations, `DECLARE` is needed in PostgreSQL.

Syntax Differences

- `FOR i IN REVERSE 1..10 LOOP` — Order must be switched for PostgreSQL.

Variables

- PL/SQL can distinguish column names and variable names.
- PL/pgSQL replaces all matching tokens by variables.
- Find a namespacing mechanism to tell apart variables, parameters, and columns.

Packages

- Use schemas to group your functions.
- Call syntax is about the same.
- But there is no equivalent public/private mechanism.

Package Variables

- Not supported by PostgreSQL
- Write a wrapper based on (temporary) tables.

Cursors

Usually, you need less cursors in PostgreSQL.

```
CURSOR foo IS SELECT ...;

BEGIN
    FOR x IN foo LOOP
```

can be simplified to

```
BEGIN
    FOR x IN SELECT ... LOOP
```

Note: The `x` is defined implicitly in Oracle. In PostgreSQL, you need to declare it.

Cursors Variables

This doesn't work in PostgreSQL:

```
CURSOR foo IS SELECT ...;
x foo%ROWTYPE;

BEGIN
    FOR x IN foo LOOP
```

Use RECORD:

```
DECLARE
    CURSOR foo IS SELECT ..;
    x RECORD;

BEGIN
    FOR x IN foo LOOP
```

PERFORM

In PostgreSQL, “procedure” calls must start with `PERFORM`. E. g.,

```
service.put_utl('Error');
```

becomes

```
PERFORM service.put_utl('Error');
```

EXECUTE

For DDL statements, `EXECUTE` might be necessary. E. g.,

```
EXECUTE 'CREATE TABLE ' || quote_ident(foo) || ...
```

Subcommits

Code that does `COMMIT` or `ROLLBACK` needs major, client-side changes. (Savepoints won't usually do the job.)

Exceptions (1)

- An exception rolls back all changes implicitly in PostgreSQL.
- You can drop most savepoint-using code from the Oracle version.
- More complex behavior needs a redesign.

Exceptions (2)

- Exception block syntax is the same.
- Exception names are different.
- Oracle supports user-defined exception names.
- Error codes are different.
- Variable SQLERRM is available.
- Of course, error messages are also different.

Exceptions (3)

We use a scheme to encode exception information into the message string:

```
RAISE name; ---> RAISE EXCEPTION 'name';
```

Similar for error codes:

```
raise_application_error(12345, 'msg');
    ---> RAISE EXCEPTION '+12345:msg';
```

Codes can be positive or negative. Write a wrapper function.

```
errcode := substr(SQLERRM, 1, 6)
```

No Data Found Exceptions

Oracle throws `NO_DATA_FOUND` exceptions for

- `SELECT`
- `INSERT`
- `UPDATE`
- `DELETE`

PostgreSQL only for:

- `SELECT INTO STRICT`

Use `IF NOT FOUND` to deal with other cases.

Logging

`dbms_output` “package” is provided by orafce. E. g.

```
dbms_output.put_line('WRONG PARAMETER: ' || par);
```

Watch for interferences from null values!

Backtraces

- orafce provides `dbms_utility.format_call_stack()`
- impossible to implement `dbms_utility.format_error_stack()` in PostgreSQL (except by patching PL/pgSQL directly)

What About PL/Java?

- Should be compatible with SQL/JRT and Oracle
- Basic functionality should work without changes
- Reality is more complex
- There is little or no experience with this scenario.

Outline

psql/sqlplus

- psql is much nicer for interactive use. :-)
- sqlplus is much nicer for scripting use. :-(
- With use of variables and naming conventions, sqlplus scripts can be converted anyway.
- Consider a wholesale rewrite.

Backup, Recovery

Build a new system using transaction log archiving or SQL dumps.

Setup, initdb

- Works completely differently.
- Forget everything you get from Oracle.
- Write new setup scripts that integrate well with the operating system.
- Forget about tablespaces, partitioning, OS tuning, etc. until you have a porting result.

JDBC

Works great, aside from SQL syntax issues

Outline

Testing

Have a test suite for:

- functions
- setup
- tables/database contents

Dividing the Work

- In PL/SQL-heavy applications, you can usually divide the work by function or package.
- Someone needs to drive and monitor integration work.
- Have a test suite.

Long-Term Maintenance

- Will the original application continue to be developed?
- . . . while the porting project runs?!?
- How is the merging going to work?
- One time ports, maintainable ports, and mergeable ports are all slightly different.

Code Formatting

- Create a code formatting standard.
- (This applies to any development project.)
- I tend to stick with the original layout.
- This is important for later updates, merges, and maintenance.

Version Control

- Use version control, even if the client doesn't.
- Prefer to use your own VCS; merge later.

Beyond the SQL

- Applications also contain setup and maintenance scripts.
- These were typically written by old-school Oracle administrators.
- Hence completely incomprehensible and written in ksh
- Half the logic usually doesn't make sense for PostgreSQL.
- Half the logic required to make PostgreSQL work will not have been written yet.

Reserve plenty of time for dealing with this.

Legacy Code and Legacy Environments

- Applications to be ported are usually very old and crufty.
- This multiplies the time required to deal with them.
- A lot of the code won't compile/run on newer operating systems.
- Half your tools won't compile/run on the old operating system.
- Everything is locked down, so you can't do anything about this.

Evaluate this carefully before starting the project.

Client Participation

- Almost always, the database clients will need to be adjusted.
- Almost always, you need someone reachable who understands the code.
- Clients think SQL code is a black box with a clean and simple interface.
- In practice, a port is like a major new software release.
- A port affects your entire system.

The client must be willing, able, and available to participate in the project.

“Josh’s Rules (of Database Contracting)”

`http://blogs.ittoolbox.com/database/soup/archives/joshs-rules-of-database-contracting-17253`

- Learn them by heart.
- Print them out.
- Post them at your office door.
- Quote them to the sales people.

Contribute Your Improvements

- Porting projects are a great source of ideas for features and bug fixes.
- Record your experiences, e. g., in the wiki.
- Contribute to orafce and ora2pg.
- Contribute to PL/pgSQL and PL/Java.

Coincidence?

If you need help:

Oracle Ask Tom: `http://asktom.oracle.com/`

PostgreSQL Ask Tom: `tgl@sss.pgh.pa.us`

Fly Way to Conclude SQL-Themed Presentation

```
COMMIT;
```