Oracle to Postgres Conversion

by James Shannon, Ben Adida, and Don Baccus

What you should know before you begin

You should know SQL relatively well. Knowing the details of Oracle SQL and Postgres SQL are obviously tremendous advantages, but the hints in this document should quickly bring you up to speed on what the differences are.

If you're porting Oracle SQL to Postgres SQL for the ACS/pg, you should also be quite familiar with AOLserver Tcl, especially the AOLserver database APIs.

In this document, we're talking about:

- Oracle 8 and 8i
- Postgres 7.0, and sometimes this also works for Postgres 6.5.3

Grammar Differences

There are a handful of grammar differences in Postgres for functionality that is actually the same. ACS/pg attempts to perform these changes automatically, leaving only the major functionality differences to be ported by hand. This is done by `db_sql_prep` which performs a number of regular expression substitutions on a piece of SQL.

Sysdate

Oracle uses the keyword `sysdate` to denote the current date and time. Postgres uses `'now'::datetime`, which ACS/pg has conveniently wrapped in a function named `sysdate()`.

ACS/pg also includes a Tcl procedure named `db_sysdate` which should be used every time the `sysdate` term appears. Thus:

```
set now [database_to_tcl_string $db "select sysdate from dual"]
```

should become

```
set now [database_to_tcl_string $db "select [db_sysdate] from dual"]
```

The Dual Table

Oracle uses the "fake" dual table for many selects. This table was created in postgres as a view to ease porting problems. This allows code to remain somewhat compatible with Oracle SQL without annoying the Postgres parser.

Sequences

Oracle's sequence grammar is `sequence_name.nextval`.
Postgres's sequence grammar is `nextval('sequence_name')`.

In Tcl, getting the next sequence value can be abstracted by calling `[db_sequence_nextval $db sequence_name]`. In case you need to include a sequence's value in a more complex SQL statement, you can use `[db_sequence_nextval_sql sequence_name]` which will return the appropriate grammar.

Decode

Oracle's handy `decode` function works as follows:

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

To evaluate this expression, Oracle compares expr to each search value one by one. If expr is equal to a search, Oracle returns the corresponding result. If no match is found, Oracle returns default, or, if default is omitted, returns null.

Postgres doesn't have the same construct. It can be replicated with:

```
CASE WHEN expr THEN expr [...] ELSE expr END
```

which returns the expression corresponding to the first true predicate. For example:

```
CASE WHEN c1 = 1 THEN 'match' ELSE 'no match' END
```

NVL

Oracle has another handy function: `NVL`. `NVL` returns its first argument if it is not null, otherwise it returns its second argument.

```
start_date := NVL(hire_date, SYSDATE);
```

The above statement will return `SYSDATE` if `hire_date` is null. Postgres has a function that performs the same thing in a more generalized way: `coalesce(expr1, expr2, expr3,....)` returns the first non-null expression that is passed to it.

Functional Differences

Postgres doesn't have all the functionality of Oracle. ACS/pg is forced to deal with these limitations with specific work-arounds. Almost everything can be done under Postgres, but some features are awaiting new versions of the open-source database.

Outer Joins

Outer Joins work as follows:

```
select a.field1, b.field2
```

```
from a, b
where a.item_id = b.item_id(+)
```

where the `(+)` indicates that, if there is no row in table `b` that matches the correct `item_id`, the match should still happen, with an empty row from table `b`. In this case, for example, for all rows in table `a` where there is no matching row in `b`, a row will still be returned where `a.field1` is correct, but `b.field2` is null.

Depending on the exact operation performed by the outer join, there are different approaches. For outer joins where specific, raw data is extracted from the outer-joined table (e.g. as above), it's best to use a `UNION` operation as follows:

```
select a.field1, b.field2
from a, b
where a.item_id = b.item_id
UNION
select a.field1, NULL as field2
from a
where 0= (select count(*) from b where b.item_id=a.item_id)
```

For queries with quadruple outer-joins, the queries can be quite long! They work quite well, though.

In certain other cases where only aggregate values are pulled out of the outer-joined table, it's possible to not use a join at all. If the original query is:

```
select a.field1, sum(b.field2)
from a, b
where a.item_id = b.item_id (+)
group by a.field1
```

then the Postgres query can look like:

```
select a.field1, b_sum_field2_by_item_id(a.item_id)
from a
```

where you've defined the function:

```
create function b_sum_field2_by_item_id(integer)
returns integer
as '
DECLARE
     v_item_id alias for $1;
BEGIN
     return sum(field2) from b where item_id= v_item_id;
END;
' language 'plpgsql';
```

Connect By

Postgres doesn't have connect by statements. Uggh. No easy way to do this.

CLOBs

Postgres doesn't have decent CLOB support. However, with the lztext extension coming with Postgres 7.0, there is no need for CLOBs in the ACS/pg. We'll be able to do everything using varchars. For now, we're using only varchar(4000).

BLOBs

Binary large object support in Postgres is very poor and unsuitable for use in a 24/7 environment, because you can't dump them with pg_dump. Backing up a database that makes use of Postgres large objects requires one to knock down the RDBMS and dump the files in the database directory.

Don Baccus put together a hack that extends AOLserver's postgres driver with BLOB-like support, by uuencoding/decoding binary files before stuffing them into or extracting them from the database. The resulting objects can be consistently dumped by "pg_dump" while the RDBMS is up and running. There is no need to interrupt service while making your backup.

To get around the one-block limit on the size of a tuple imposed by Postgres, the driver segments the encoded data into 8K chunks.

Postgres large objects are scheduled for a major overhaul in summer 2000. Because of this, only the BLOB functionality used by the ACS was implemented.

To use the BLOB driver extension, you must first create a column of type "integer" with the name "lob" in the table that will store the BLOB, and a trigger on it that calls "on_lob_ref". You **must** use the name "lob". Here's an example:

```
create table my_table (
    my_key                      integer primary key,
        lob                             integer references lobs,
    my_other_data       some_type -- etc
);

create trigger my_table_lob_trig before insert or delete or update
on my_table for each row execute procedure on_lob_ref();
```

To put a binary file into "my_table":

```
set lob [database_to_tcl_string $db "select empty_lob()"]

ns_db dml $db "begin"
ns_db dml $db "update my_table set lob = $lob where my_key = $my_key"
ns_pg blob_dml_file $db $lob $tmp_filename
ns_db dml $db "end"
```

Note that the call to ns_pg to stuff the file into the database MUST be wrapped in a transaction, even if

you're not updating any other tables at the same time. The driver will return an error if you don't.

To return a large object stored in "my_table" to the user:

```
set lob [database_to_tcl_string $db "select lob from my_table
                                     where my_key = $my_key"]
ns_pg blob_write $db $lob
```

Note that you don't need to wrap the call to blob_write in a transaction, as the database isn't being modified.

The large objects are automatically deleted when no longer used. To replace the large object stored in an existing record, just allocate a new one by calling "empty_lob()" and assign the returned key to the "lob" column in your table.

---

james@motifstudios.com / ben@adida.net