

Oracle to PostgreSQL beyond the Syntax

When DBMS Design Differences Matter

PGCONF.EU
2026

Tino Engelbrecht
Joshua Steinmann

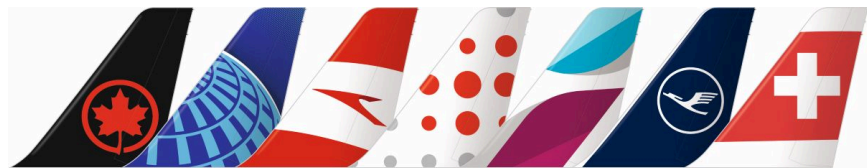
Overview

1. Background
2. (Easy) Replacements
3. Fun With Numbers
4. Rethinking Updates
5. Key Takeaways

Background

Who we are

- Software Developers, Ops, “own DBAs”
- Department of Revenue Accounting
- Customers are some of the largest airlines worldwide



Atlantic Joint Venture

Our experience

- In 2025: Migration project from Oracle to Postgres
- Learned a lot from attendance at Postgres conferences in Germany and Europe



01 (Easy) Replacements

Easy replacements

```
(SUBSTR(import_month, 1, 4) + 1 || '00') + NVL(late_coming_months, 0)
```

Easy replacements

```
(SUBSTR(import_month, 1, 4) + 1 || '00') + NVL(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE

Easy replacements

```
(SUBSTR(import_month, 1, 4) + 1 || '00') + COALESCE(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE

Easy replacements

```
(SUBSTR(import_month, 1, 4) + 1 || '00') + COALESCE(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE
- Postgres is less forgiving about types: adding 1 to a text needs a cast

Easy replacements

```
(SUBSTR(import_month, 1, 4)::BIGINT + 1 || '00') + COALESCE(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE
- Postgres is less forgiving about types: adding 1 to a text needs a cast

Easy replacements

```
(SUBSTR(import_month, 1, 4)::BIGINT + 1 || '00') + COALESCE(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE
- Postgres is less forgiving about types: adding 1 to a text needs a cast
- Oracle accepts some crimes thanks to implicit type conversions – there are better ways

Easy replacements

```
((SUBSTR(import_month, 1, 4)::BIGINT + 1) * 100) + COALESCE(late_coming_months, 0)
```

- Some functions just have a different name: NVL → COALESCE
- Postgres is less forgiving about types: adding 1 to a text needs a cast
- Oracle accepts some crimes thanks to implicit type conversions – there are better ways

How to translate **DECODE** ?

```
-- Oracle  
SELECT DECODE(n, null, 'NULL', 1, 'One', 'Other')  
FROM nums;
```

- compare **n** with each search value
- return the result of the first match
- last argument is the default

n	result
1	'One'
2	'Other'
null	'NULL'

How to translate **DECODE** ?

```
-- Oracle
SELECT DECODE(n, null, 'NULL', 1, 'One', 'Other')
FROM nums;
```

n	result
1	'One'
2	'Other'
null	'NULL'

How to translate **DECODE** ?

```
-- Oracle
SELECT DECODE(n, null, 'NULL', 1, 'One', 'Other')
FROM nums;
```

```
SELECT CASE n
  WHEN null::INTEGER THEN 'NULL'
  WHEN 1 THEN 'One'
  ELSE 'Other' END
FROM nums;
```

n	result
1	'One'
2	'Other'
null	'Other'

`null = null` is not `true` - it is `null`.

How to translate **DECODE** ?

```
-- Oracle
SELECT DECODE(n, null, 'NULL', 1, 'One', 'Other')
FROM nums;
```

```
SELECT CASE n
  WHEN null::INTEGER THEN 'NULL'
  WHEN 1 THEN 'One'
  ELSE 'Other' END
FROM nums;
```

```
SELECT CASE
  WHEN n IS NULL THEN 'NULL'
  WHEN n = 1 THEN 'One'
  ELSE 'Other' END
FROM nums;
```

`null = null` is not `true` - it is `null`.

n	result
1	'One'
2	'Other'
null	'NULL'

Join on **DECODE**

```
-- Oracle
SELECT a.n, b.n FROM nums a JOIN nums b
      ON DECODE(a.n, b.n, 1, 0) = 1;
```

```
SELECT a.n, b.n FROM nums a JOIN nums b
      ON CASE
         WHEN a.n IS NULL AND b.n IS NULL THEN 1
         WHEN a.n = b.n THEN 1
         ELSE 0 END = 1;
```

```
SELECT a.n, b.n FROM nums a JOIN nums b
      ON a.n IS NOT DISTINCT FROM b.n;
```

a.n	b.n
1	1
2	2
null	null

Correct – but imagine large views with many joins.

Is there a shorter form?

IS NOT DISTINCT FROM

Expression	Result
1 IS NOT DISTINCT FROM 1	true
1 IS NOT DISTINCT FROM 2	false
1 IS NOT DISTINCT FROM NULL	false
NULL IS NOT DISTINCT FROM NULL	true

```
SELECT * FROM tbl WHERE a IS NOT DISTINCT FROM 1
-- can be rewritten to
SELECT * FROM tbl WHERE a = 1
```

```
SELECT * FROM tbl WHERE a IS NOT DISTINCT FROM NULL
-- rewritten to
SELECT * FROM tbl WHERE a IS NULL
```

```
SELECT * FROM tbl_one LEFT JOIN tbl_two
ON tbl_one.a IS NOT DISTINCT FROM tbl_two.a
```

In joins: no index usage.

Workaround with functions

```
SELECT a.n, b.n FROM nums a JOIN nums b  
ON COALESCE(a.n::text, '||NULL||')  
= COALESCE(b.n::text, '||NULL||');
```

```
CREATE INDEX idx_nums_coalesce  
ON nums (COALESCE(n::text, '||NULL||'));
```

a.n	b.n
1	1
2	2
null	null

Same name, different behavior: CURRENT_DATE

Oracle

```
ALTER SESSION SET  
  NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';  
SELECT CURRENT_DATE FROM DUAL;
```

```
2026-10-16 15:30:00
```

Oracle's default `NLS_DATE_FORMAT` (`DD-MON-RR`)
hides the time, but the value always includes it.

PostgreSQL

```
SELECT CURRENT_DATE;
```

```
2026-10-16
```

PostgreSQL `CURRENT_DATE` is a date only, so using it
where the Oracle value's time mattered changes the
result.

Same name, different type—and a different clock

Oracle

All below use current time, not transaction start.

- **Session zone:** `CURRENT_DATE` , `CURRENT_TIMESTAMP`
- **Server OS zone:** `SYSDATE` , `SYSTIMESTAMP`

`CURRENT_DATE` / `SYSDATE` → `DATE` : date + time to seconds, **no stored zone**.

`CURRENT_TIMESTAMP` / `SYSTIMESTAMP` → `TIMESTAMP`
`WITH TIME ZONE` : fractional seconds + zone.

PostgreSQL

- **Transaction start:** `CURRENT_DATE` , `CURRENT_TIMESTAMP`
/ `now()`
- **Statement start:** `statement_timestamp()`
- **Current time:** `clock_timestamp()` (changes within a statement)

All use session `TimeZone` for the date / display.

`CURRENT_DATE` → `date` : date only, **no stored zone**.

The others → `timestamptz` : UTC instant, displayed in session zone; original zone not retained.

Session zone ≠ session start. None of these functions uses session start time.

02 Fun with numbers

Exact numeric types

Oracle

- Only one data type: `NUMBER(precision, scale)`
 - stored as up to 20 base-100 digits → max precision of 39 or 40 digits, depending on the position of the decimal point
 - `INT`, `SMALLINT`, `INTEGER` are aliases for `NUMBER(38,0)`
- Always performs “normal” decimal division
- Always returns max precision as result of arithmetic operations

Postgres

- Proper `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC` (same as `DECIMAL`)
- Performs integer division on integer types
- Max precision is 1000
- Inexact operations (division, `sqrt`, ...) return at least 16 significant digits
 - scale of the result is at least as big as the scale of both operands

03 Rethinking Updates

Two transaction systems

Oracle – undo

- Row is changed **in place**; the before image goes to an **undo segment**
- Undo serves rollback, read consistency
- **COMMIT** : fast, independent of transaction size
- **ROLLBACK** : re-applies undo records (cost scales with changed data)

Postgres 18 – MVCC

- Update = **new row version** + old one gets outdated
- No undo: old versions stay in the table for concurrent readers
- **COMMIT** and **ROLLBACK** : just flip the xid status
- **dead tuples** that **VACUUM** has to reclaim

Oracle updates

```
UPDATE records  
  SET reporting_month = 'Apr '26'  
 WHERE update_flag;
```

id	reporting_month	update_flag
1	Jan '26	
2	Feb '26	
3	Mar '26	✓
4	Mar '26	✓
5	Mar '26	✓
6	Mar '26	

Oracle updates

```
UPDATE records  
  SET reporting_month = 'Apr '26'  
 WHERE update_flag;
```

id	reporting_month	update_flag
1	Jan '26	
2	Feb '26	
3	Apr '26	✓
4	Apr '26	✓
5	Apr '26	✓
6	Mar '26	

undo

id	reporting_month
3	Mar '26
4	Mar '26
5	Mar '26

Postgres updates

```
UPDATE records  
  SET reporting_month = 'Apr '26'  
 WHERE update_flag;
```

id	reporting_month	update_flag	xmin	xmax
1	Jan '26		100	0
2	Feb '26		200	0
3	Mar '26	✓	300	0
4	Mar '26	✓	400	0
5	Mar '26	✓	500	0
6	Mar '26		600	0

Postgres updates

```
UPDATE records
  SET reporting_month = 'Apr '26'
 WHERE update_flag;
```

Dead tuples: table size ↑ index size ↑

id	reporting_month	update_flag	xmin	xmax
1	Jan '26		100	0
2	Feb '26		200	0
3	Mar '26	✓	300	700
4	Mar '26	✓	400	700
5	Mar '26	✓	500	700
6	Mar '26		600	0
3	Apr '26	✓	700	0
4	Apr '26	✓	700	0
5	Apr '26	✓	700	0

Postgres Vacuum

```
UPDATE records
  SET reporting_month = 'Apr '26'
  WHERE update_flag;
```

```
VACUUM records;
```



Reuse the free space across many pages or repack the table.

Both: additional workload and table locks.

id	reporting_month	update_flag	xmin	xmax
1	Jan '26		100	0
2	Feb '26		200	0
6	Mar '26		600	0
3	Apr '26	✓	700	0
4	Apr '26	✓	700	0
5	Apr '26	✓	700	0

Business logic change

```
UPDATE records  
  SET reporting_month = 'Mar '26'  
  WHERE update_flag;
```

id	reporting_month	update_flag	xmin	xmax
1	Jan '26		100	0
2	Feb '26		200	0
3	<null>		300	0
4	<null>		400	0
5	<null>		500	0
6	<null>	✓	600	0

Business logic change

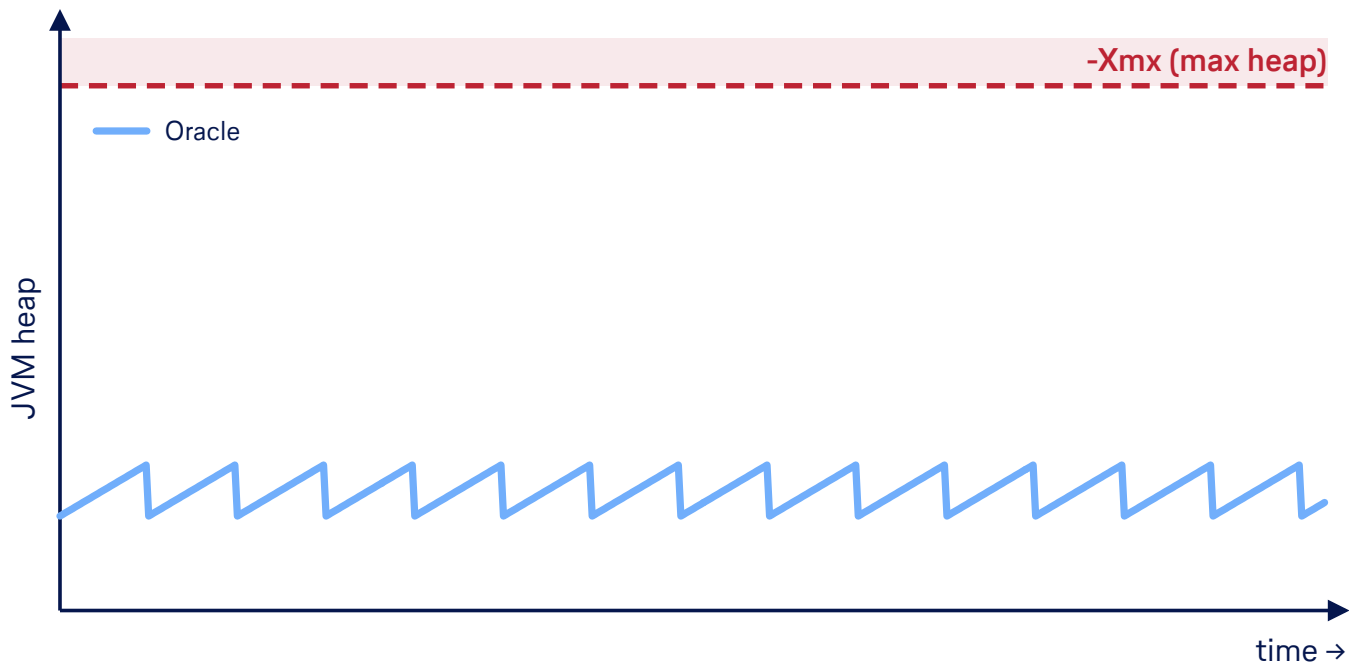
```
UPDATE records
  SET reporting_month = 'Mar '26'
 WHERE update_flag;
```

Only update what is necessary: less bloat, better performance.

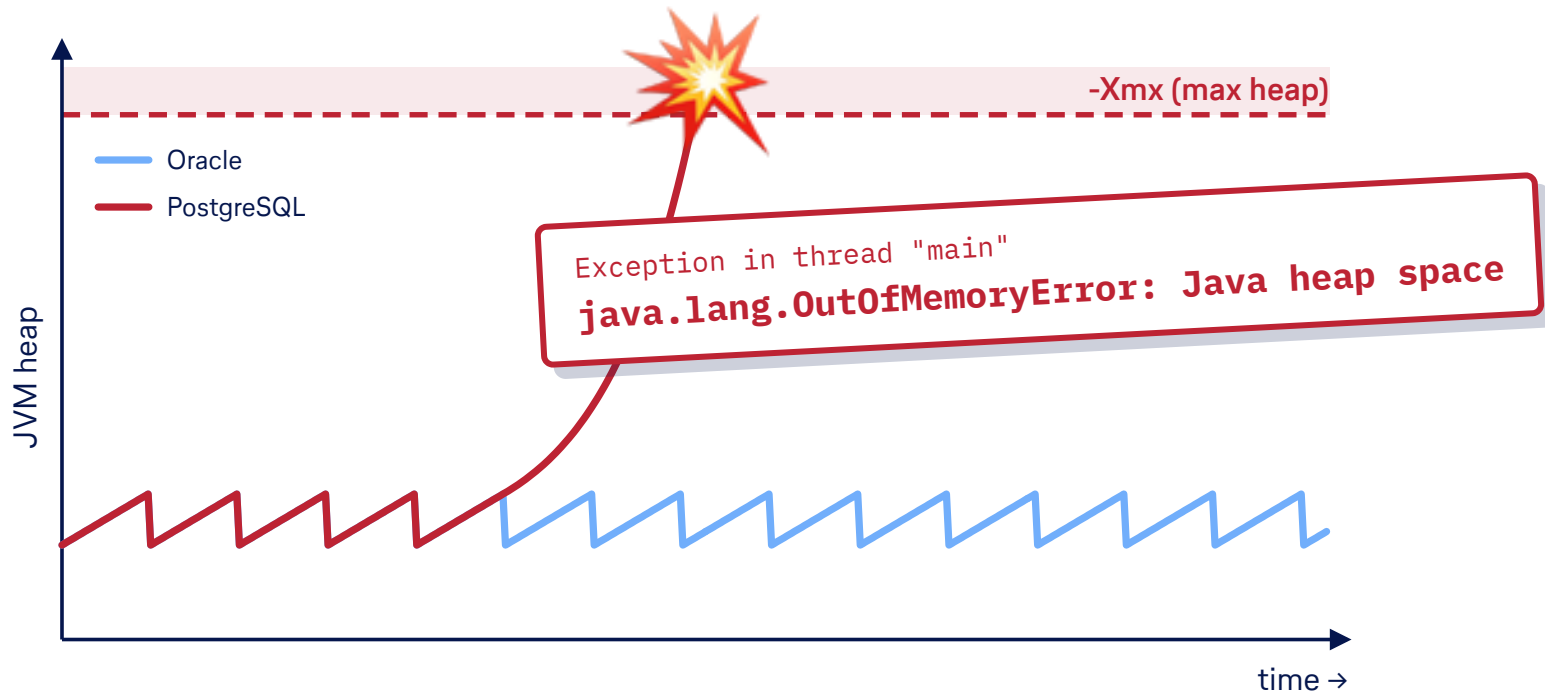
id	reporting_month	update_flag	xmin	xmax
1	Jan '26		100	0
2	Feb '26		200	0
3	<null>		300	0
4	<null>		400	0
5	<null>		500	0
6	<null>	✓	600	700
6	Mar '26	✓	700	0

04 Let's go to production

What could go wrong now?



What could go wrong now?



A cursor reader that reads everything?

Oracle

- JDBC can fetch rows in batches even with autocommit enabled
- Spring Batch's `JpaCursorItemReader` appears to stream just fine

PostgreSQL / pgJDBC

- Cursor fetching needs an **open transaction** on the reader's connection
- Also needs a **positive fetch size** and a forward-only result set
- Otherwise: driver buffers the **entire result set**

Same reader, different memory usage: application heap exhausted → OOM → application crash.

Our fix: a custom cursor item reader that opens a transaction before reading and closes it after all rows are read.

Key takeaways for migrating

- 1 Make it run on Postgres ASAP
- 2 Find mistakes in translation, ensure correctness
- 3 Set goals for performance (benchmark old system?)
- 4 Optimize performance – don't shy away from hacks!
- 5 Find systematic issues
- 6 Go back to 4 until performance goals are met



There are bad practices in your system, you just got away with it.

PostgreSQL User Group Hamburg



Scan to find out more and join us.

Thank you for your attention!

Questions?

