Oracle vs Postgres

This page is the repository of the differences between Postgresql and Oracle.

They fall into 3 categories

1. Differences in features (semantics as well as implementation)
2. Differences in query language
3. Differences and alternatives for HA, Backup and Recovery, Connection pooling

Differences in Features

1. Temp Table

Temporary tables are created using sessions and used during the session. In Oracle, they are created automatically. In Postgresql they have to be created for every session and they are not visible outside the session. In Oracle, Temp tables can be accessed outside the sessions.

Info on Postgresql temp table creation can be found at
http://www.postgresql.org/docs/current/static/sql-createtable.html

> Temporary tables exist in a special schema, so a schema name cannot be given when creating a temporary table. The name of the table must be distinct from the name of any other table, sequence, index, view, or foreign table in the same schema.

From http://www.postgresql.org/docs/current/static/sql-createtable.html#SQL-CREATETABLE-COMPATIBILITY

```
TEMPORARY or TEMP
    If specified, the table is created as a
temporary table. Temporary tables are automatically dropped at the end
of a session, or optionally at the end of the current transaction (see
ON COMMIT below). Existing permanent tables with the same name are not
visible to the current session while the temporary table exists, unless
they are referenced with schema-qualified names. Any indexes created on a
 temporary table are automatically temporary as well.
    The
autovacuum daemon cannot access and therefore cannot vacuum or analyze
temporary tables. For this reason, appropriate vacuum and analyze
operations should be performed via session SQL commands. For example, if
 a temporary table is going to be used in complex queries, it is wise to
 run ANALYZE on the temporary table after it is populated.

Optionally, GLOBAL or LOCAL can be written before TEMPORARY or TEMP.
This presently makes no difference in PostgreSQL and is deprecated; see
Compatibility.
```

2. Partitioning

Currently Blue hornet is using two types of partitioning

Normal partitions + Range Partitions + Hash partitions

Postgres differs from Oracle partitions in the following respects

1. Normal partitions are actually separate tables that hang off a meta table that contains the definition but contains no data.
2. Individual tables normally do not add columns. Individual tables / partitions can have their own indexes.
3. Partitions need to be pre-created for data that will be inserted. For example if data is partitioned by day then there will be need to create empty tables for them before hand. There are various hacks using triggers and other ways to autocreate them but they are non-standard and not recommended.
4. Postgresql does not support Hash partitions at this point in time.
5. It is possible to attach and detach partitions in Postgresql. This is really useful in archival. It does not seem possible at this point in time to compress them when archiving.
6. If someone tries to insert data into the meta/parent table, it is possible to have a trigger to insert it into the right partition but this has overheads.
7. When creating check constraints it is not possible to check if they are mutually exclusive. So this could lead to inconsistencies if not done properly.

More information at http://www.postgresql.org/docs/current/static/ddl-partitioning.html

3. Indexes

PostgreSQL has different types of indexes specialized for different operations See http://www.postgresql.org/docs/9.1/static/indexes.html

See also => https://devcenter.heroku.com/articles/postgresql-indexes

Types of Indexes

1. **B-tree**

B-tree indexes can also be used to retrieve data in sorted order.
Also only B-tree indexes would be unique

2. Hash

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator.

```
Hash index operations are not presently WAL-logged, so hash indexes
might need to be rebuilt with REINDEX after a database crash if there
were unwritten changes. Also, changes to hash indexes are not replicated
 over streaming or file-based replication after the initial base backup,
 so they give wrong answers to queries that subsequently use them. For
these reasons, hash index use is presently discouraged.
```

3. GiST
http://www.postgresql.org/docs/current/static/gist-intro.html
http://www.sai.msu.su/~megera/wiki/GiST
GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST

4. SP-GiST
http://www.sai.msu.su/~megera/wiki/spgist_dev
http://www.postgresql.org/docs/current/static/spgist.html
SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries).

5. GIN
GIN stands for Generalized Inverted Index. GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words.
http://www.sai.msu.su/~megera/wiki/Gin
http://www.postgresql.org/docs/current/static/gin.html

```
Create vs. insert
    Insertion into a GIN index can be slow due
to the likelihood of many keys being inserted for each item. So, for
bulk insertions into a table it is advisable to drop the GIN index and
recreate it after finishing bulk insertion.
```

6. BRIN
BRIN indexes (a shorthand for Block Range indexes) store summaries about the values stored in consecutive table physical block ranges. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range

BRIN is designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table. A block range is a group of pages that are physically adjacent in the table; for each block range, some summary info is stored by the index. For example, a table storing a store's sale orders might have a date column on which each order was placed, and most of the time the entries for earlier orders will appear earlier in the table as well; a table storing a ZIP code column might have all codes for a city grouped together naturally.

Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (SELECT statements) to occur on the table in parallel with index creation, but writes (INSERT, UPDATE, DELETE) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see Building Indexes Concurrently.

http://www.postgresql.org/docs/current/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

4. External Tables

There is no analogue to External tables in Postgresql. The nearest way to read data from external files is provided by the Foreign Data Wrapper module. We will need to write a FDW module to read from external files to the database. This is non-trivial to write and test.

https://docs.oracle.com/cd/B19306_01/server.102/b14215/et_concepts.htm
http://allthingsoracle.com/external-tables-an-introduction/

http://www.postgresql.org/docs/current/static/postgres-fdw.html
https://wiki.postgresql.org/wiki/Foreign_data_wrappers
https://wiki.postgresql.org/images/7/7e/Conferences-write_a_foreign_data_wrapper_in_15_minutes-presentation.pdf
http://pgxn.org/tag/fdw/

5. Query planning and tuning

The query plan can be seen by using the Explain command. The Explain Analyze command actually runs the query and give the time spent on each step required to get the result. Further reading at

https://wiki.postgresql.org/wiki/Using_EXPLAIN

http://postgresguide.com/performance/explain.html

Also it is possible to tune queries at the global level using configuration or by using per-session variables but it is not possible to tune the queries by matching a certain query profile (like in the case of Oracle. There are also tools like privacy badger and system tables like pg_stat_* that can be used to tune and find long running / inefficient queries.

6. Arrays

It is possible to store JSON data in PostgreSQL. It is possible to index it as well using GIN indexes to make them searchable There are two ways of storing JSON data. One is to use the data type JSON, the other is to use the datatype JSONB, JSON stores the json document as is and order is preserved but it is slow. JSONB (or JSON binary) puts the data in binary format but does not preserve order. It is faster than JSON storage though.

Further reading:

http://www.postgresql.org/docs/9.5/static/datatype-json.html

http://www.postgresql.org/docs/9.5/static/functions-json.html

https://www.compose.io/articles/is-postgresql-your-next-json-database/

7. Materialised views

It is easy to create materialized views in PostgreSQL similar to Oracle and the syntax is fairly straightforward. The query is tored with the view and it can be easily refreshed (full refresh).

http://www.postgresql.org/docs/current/static/rules-materializedviews.html

Partial refreshes are not natively supported but there are hack for this workaround => http://dba.stackexchange.com/questions/86779/refresh-materalized-view-incrementally-in-postgresql

Since materialized views are proper tables under PostgreSQL which can have their own indexes it is possible to generate a table with specific syntax to regenerate the table partly

8. Defragmentation

PostgreSQL needs to be regularly defragmented to keep query performance and disk usage sane. The command used for defragmentation is Vacuum. It is possible to do a full or partial vacuum. Also it is recommended to keep autovacuum on so this process is regularly done automatically.

Further reading at http://www.postgresql.org/docs/current/static/sql-vacuum.html and http://www.postgresql.org/docs/9.2/static/routine-vacuuming.html

9. Transaction Log

PostgreSQL has a transaction log similar to redo log of Oracle. It is helps in checkpointing and crash recovery. also it can be used in log shipping and synchronous replication to a slave server. It can also be used in logical decoding and replaying only specific transactions. It can also be sued to restore the database to a specific time in the past.

Further reading at http://www.postgresql.org/docs/current/static/wal.htm

10. Blob and large object storage

http://giswiki.hsr.ch/PostgreSQL_-_Binary_Large_Objects

https://wiki.postgresql.org/wiki/BinaryFilesInDB

http://www.postgresql.org/docs/current/static/largeobjects.html

http://www.postgresql.org/docs/current/static/storage-toast.html

http://www.postgresql.org/docs/9.1/static/datatype-binary.html

The maximum limit for large files is 1GB.

11. Merge / Upserts

Upserts are basically update if already present or insert if absent. Oracle uses "merge" command to achieve the same effect. PostgreSQL extends the insert syntax but the conflict management differs subtly between the two of them. If not handled with proper clauses there can be unintended side effects.

some examples

http://stackoverflow.com/questions/237327/oracle-how-to-upsert-update-or-insert-into-a-table

http://coderedux.com/on-postgresql/upsert-bad-apples

12. Triggers

Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire.

More specific at

http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/triggers.htm#LNPLS723

http://www.tutorialspoint.com/postgresql/postgresql_triggers.htm

Differences in Query Language

1. Dummy Table

Oracle's pseudo table/view DUAL does not exist on PostgreSql. Oracle syntax requires ' from ...' to consider a valid statement. PostgreSQL does not need ' from ...', just use 'select ...' without from.

Oracle: SELECT 'X' FROM dual;

Postgres: SELECT 'X';

http://www.postgresql.org/docs/current/static/sql-select.html

2. Current System Date

Oracle's built-in function SYSDATE does *not* work in PostgreSQL. The ANSI standard defines CURRENT_DATE or CURRENT_TIMESTAMP which is supported by Postgres.

Oracle:

SELECT TRUNC(SYSDATE) FROM dual; -- 'dd/mm/yyyy'

SELECT SYSDATE FROM dual; -- 'dd/mm/yyyy hh24:mm:ssss'

PostgreSQL:

SELECT CURRENT_DATE; -- 'dd/mm/yyyy'

SELECT CURRENT_TIMESTAMP; -- 'dd/mm/yyyy hh24:mm:ssss'

http://www.postgresql.org/docs/current/static/functions-datetime.html

3. Sequences

Oracle's sequence grammar is sequence_name.nextval.

PostgreSQL's sequence grammar is nextval('sequence_name').

Oracle: select sequence_name.nextval from dual;

Postgres: select nextval('sequence_name')

http://www.postgresql.org/docs/9.5/static/sql-createsequence.html

4. Decode

Oracle's DECODE( ) function does *not* work in PostgreSQL, you need to substitute DECODE( ) for CASE ... WHEN ... THEN ... ELSE ... END

Oracle:

```
SELECT DECODE( any_column,
            1, 'one',
            2, 'two',
              'others'
          )
   FROM ANY_TABLE;
```

Postgres:

```
SELECT CASE WHEN any_column = 1 THEN 'one'
          WHEN any_column = 2 THEN 'two'
          ELSE 'others'
            END
     FROM ANY_TABLE;
```

http://www.postgresql.org/docs/current/static/functions-conditional.html

5. NVL

Oracle null value conversion NVL( ) does *not* work in PostgreSQL, you need to use COALESCE(expr#1, expr#2, ..., expr#n)

Oracle: SELECT NVL(any_column, 0) FROM ANY_TABLE;

Postgres: SELECT COALESCE(any_column, 0) FROM ANY_TABLE;

http://www.postgresql.org/docs/current/static/functions-conditional.html

6.Subquery in FROM

PostgreSQL requires a sub-SELECT surrounded by parentheses, and an alias must be provided for it. The alias is not mandatory for Oracle.

Oracle: SELECT * FROM (SELECT * FROM table_a)

Postgres: SELECT * FROM (SELECT * FROM table_a) as foo

7.Outer Joins

Outer Joins in Oracle work as follows:

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

In Postgresql: The query should be in ANSI standard format.

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

```
from a
left outer join b
on a.item_id = b.item_id;
```

http://www.postgresql.org/docs/9.5/interactive/sql-select.html

8.Hierarchical Queries

Being Oracle SQL extension, CONNECT BY is not available in PostgreSQL. PostgreSQL implements Common Table Expressions (CTE), SQL-standard way of dealing with hierarchical data.

http://www.postgresql.org/docs/current/interactive/queries-with.html

9. NO_DATA_FOUND and TOO_MANY_ROWS

These exceptions are disabled by default for selects in PLpgSQL. You need to add keyword STRICT after any keyword INTO in all selects, when You need to keep single row checking in stored PLpgSQL code.

http://www.postgresql.org/docs/9.5/interactive/plpgsql-statements.html

10. Limiting Result Sets

Oracle -> ROWNUM

PostgreSQL -> LIMIT {OFFSET} or FETCH FIRST <n> ROWS ONLY

http://www.postgresql.org/docs/9.5/static/sql-select.html#SQL-LIMIT

11. Automatic Key Generation

PostgreSQL's best offering for a column with auto-generated values is to declare a column of 'type' SERIAL:

```
CREATE TABLE tablename (
 tablename_id SERIAL,
 ...
)
```

'SERIAL' is a shorthand for creating a sequence and using that sequence to create unique integers for a column. If the table is dropped, PostgreSQL will drop the sequence which was created as a side-effect of using the SERIAL type.

If you want an auto-incrementing column in Oracle, then create a sequence and use that sequence in a trigger associated to the table.

http://www.postgresql.org/docs/9.5/static/datatype-numeric.html