Analyzing Spatial Query Performance Improvements in Oracle Spatial 12c

Through

Cross-Vendor Comparison

Mr Simon G Greener

September 2014

www.spatialdbadvisor.com

Report Table Of Contents

1. Executive Summary

This analysis was carried out to discover how the enhancements made in Oracle Spatial and Graph 12c (hereafter referred to simply as **Oracle Spatial 12c)** have affected the performance of Spatial functionality relative to a number of popular alternative approaches. While it is not intended to be a comprehensive analysis, it is intended to give a better understanding of the current performance of the spatial operators and functions in Oracle Spatial 12c. Over the years, there have been claims and counter-claims regarding relative performance of spatial database technologies; this analysis attempts to address this issue in a systematic manner.

These tests measured single user (no parallel operations) with mixed (cold and hot) cache performance. This is because only Oracle Spatial 12c can perform parallel operations and we wished to show an "apples to apples" comparison. Since many prior tests demonstrate that parallel operations enable near linear performance improvements, it is safe to say that when operations are parallel-enabled the Oracle Spatial results would demonstrate similar near-linear improvements.

All tests performed use Oracle Database 12c with Spatial option; the 12c SPATIAL_VECTOR_ACCELERATION parameter is invoked (TRUE).

While a larger set of tests were considered, the relative capabilities of the comparison data types reduced the set to the following:

- Primary Filtering;
- Secondary Filtering with simple masks (ANYINTERACT, INSIDE, CONTAINS)
- Aggregation;
- Distance computations;
- Nearest neighbour.

The following tests of functionality and performance were not attempted.

- Union/Intersection operations between two objects;
- Data loading and index creation;
- Queries with multiple predicates (one spatial and one or more non-spatial);
- Spatial joins between two or more tables.

The following test was attempted but limitations in the non-Oracle products made fair comparison difficult.

- Relation operations (i.e. determining spatial relationship between two objects).

1.1. Summary of Results

Overall, this analysis has shown that the performance of *all the Oracle Spatial 12c functions tested with spatial vector acceleration invoked* is as good as, equal to, or better than, its competitors (open source or commercial).

In the desire to perform "apples to apples" testing, there are a number of capabilities in Oracle Spatial 12c that were not addressed:

- Since neither PostGIS nor the 3rd-party type support parallel operations, no Oracle Spatial operations in this testing were performed in parallel. Since virtually all Oracle Spatial operators are parallel-enabled which have been shown to scale in a near-linear fashion, results for Oracle Spatial 12c would be multiple times faster when exploiting parallel architectures.

- Of the technologies tested, only Oracle Spatial and PostGIS were capable of performing geodetic operations. PostGIS uses a separate data type for geodetic operations that supports fewer operations than its geometry type. This analysis has shown that Oracle's functional offerings and simplicity of use for geodetic data are superior to the analysed competitors. Simplicity of use and *functional coverage* are just as important as straight performance.
- Most of the geometry type pairs (polygon->polygon; polygon->line) were attempted but point within polygon (State, BlockGroup and Building) was not attempted due to a lack of time. This was a major oversight of the testing as this did not give Oracle Spatial 12c the chance to show its optimized point-in-polygon functionality which allows it to resolve all point-in-polygon relationships via Rtree operations (not just the MBR relationships as in primary filtering).
- Only Oracle Spatial 12c and PostGIS have the CoveredBy and Covers secondary operators; as such these are not tested and reported.
- Oracle has a form of a precision model, the metadata that describes how two coordinates are to be compared for equality; the other two do not. This model is exposed through the metadata it stores and the use of that metadata in functions.

The results of these test revealed:

- Primary filter geometry results showed negligible performance variations for most tests
  - All within 5/1000 sec per search for polygons.
  - All within 8/1000 sec per search for points.
  - PostGIS and Oracle Spatial were within 7/1000 sec per search for linestrings; the 3rd party type was an order of magnitude slower.
- Primary filter geodetic results comparing Oracle Spatial to PostGIS showed nominally better search times for PostGIS, similar throughput results for both products in linestring tests, but nearly 4 to 5 times better throughput for Oracle Spatial in the polygon and point tests.
- Secondary filter geometry results generally showed significantly better performance for Oracle Spatial across all tests.
- Secondary filter geodetic tests showed Oracle Spatial performance to be 30 to nearly 350 times faster than PostGIS.
- Aggregation tests on both projected and geodetic data showed Oracle Spatial to be 5 to 8 times faster than PostGIS, but both to have acceptable performance.
- The SDO_BUFFER tests performed show that the Oracle Spatial Java implementation performs substantially slower than the PostGIS and 3rd Party implementations. While the relative difference of this function is significant compared to its competitors, the overall performance is no better than *usable* when executed individually. Where larger scale usage is involved, spatial practitioners prefer *fast* to *usable* so one might hope that Oracle recodes this operation in C++. One should note that in practice this single processor performance difference can be mitigated substantially by running Buffer operations using Oracle Database parallel query.

2. Introduction

One's understanding of the performance and functionality of Oracle Spatial often depends on one's prior experience:

1. Those who are significant users of the product (and who know where the real issues are) and,
2. Those who use other products and often have a lesser level of direct knowledge (or one filtered by vendor sales and marketing).

Actual users of the product know first hand that performance before the 12c release was more than acceptable for *most* operations; the performance was also known to be insufficient for a small number of operators, in particular polygon aggregation (SDO_AGGR_UNION). For those who use other products, much of what they consider to be deficiencies in Oracle Spatial 12c when compared to non-Oracle implementations is mainly anecdotal (c.f., those who think the Oracle database is difficult to use, administer or tune without having any experience of the product) or based on familiarity with other offerings (we *know* our product is better than Oracle's).

Much of the criticism (from both sides) has never been objectively examined or tested, until now.

Oracle has invested heavily in its spatial type since its first release in 1997. As with other vendors, Oracle has continuously worked on the performance of its offering e.g., optimised RTree indexing and improvements in secondary operator performance. Oracle Spatial 12c is different from previous releases in that a sustained and systematic examination of all aspects of the product's performance was undertaken and, where possible, improved. The 12c release is proof of that ongoing dedication to performance improvement.

This study does not attempt to compare Oracle Spatial 12c with previous releases: others have undertaken that task. This study is also not about determining whether Oracle's SDO_GEOMETRY type is objectively faster than its competitors; rather it is to dispel real or claimed performance concerns experienced by real users or expressed by competitors. This can only be done by comparing Spatial against competitors that either have a recognised (benchmark) level of performance by practitioners, or which have a reputation for quality and performance.

Before conducting the comparison, the technical differences that make a completely statistically accurate comparison challenging to perform were identified and exposed. For example, different data storage methods, or the presence or lack of a precision model, may or may not make a contribution to performance. Controlling the many variables involved in a rigorous comparison is a time consuming task, and isn't always practical.

2.1. Acknowledgements

Publication of performance benchmarks involving Oracle software are something that is constrained by license agreements (see below). This comparison has been sanctioned by Oracle and for this I am thankful to them for allowing for the compilation and publication of this comparison.

2.2. Licensing Issues

The Oracle developer software license includes this clause:

> "You may not:
> - ….
> - disclose results of any program benchmark tests without our prior consent."

Consent to both conduct and publish the results of this comparison was provided by Oracle.

PostGIS, as an open source product, does not impose any license restrictions on performance benchmarking.

At the time of the comparison, the 3rd party spatial type provider's developer license only restricted benchmarking for beta versions of its software. The name of the actual vendor whose spatial type was used in this comparison has been suppressed; additionally, specific schema or method names that could identify the vendor have been obfuscated.

2.3. Hardware and Software

Operating System:		Oracle Linux 6 – x86 64 bit
Hardware:		16 CPUs (64 cores) - Intel Xeon Processor E5620 48GB RAM Samsung Electronics 840 EVO-Series 500GB 2.5-Inch SATA III SSD
Database Software:		Oracle Database 12.1.0.1 64 bit PostgreSQL 9.3 64 bit
Spatial Software	Oracle	Oracle Spatial and Graph 12.1.0.1 – Spatial Vector Acceleration enabled
	Commercial Vendor	3rd party ST_GEOMETRY 64bit EXTPROC (Latest production release as at time of testing)
	Open Source	PostGIS 2.1.3

2.4. Approach

The aim of this comparison is to look at spatial structure and algorithm efficiency; it is not meant to be a comparison between the Oracle database and the PostgreSQL database. Many tests are constructed to minimize or remove database caching effects e.g., buffering, distance calculations.

2.4.1. Cold, Hot and Hardware

Comparing "cold" databases vs "hot" databases is considered to be a test more of one database vs. another. When testing cold/hot states the aim is mainly to try and remove variables that may affect the comparison but are part of the comparison. How much the cold/hot paradigm affects this particular comparison is debatable. The hardware specification used for this comparison affects both the hot/cold issue and also the performance of each database: 48G of RAM for databases that were much smaller than this meant that much of the tables ended in RAM pretty quickly. Additionally, the hardware had no traditional spindle disks: all the data was held on very fast SATA III SSD. With both of these, the concept of "cold" becomes questionable. Because of all these issues, the databases were allowed to run as they were (which is more akin to what a "normal" database might be like: a collection of cold and hot data).

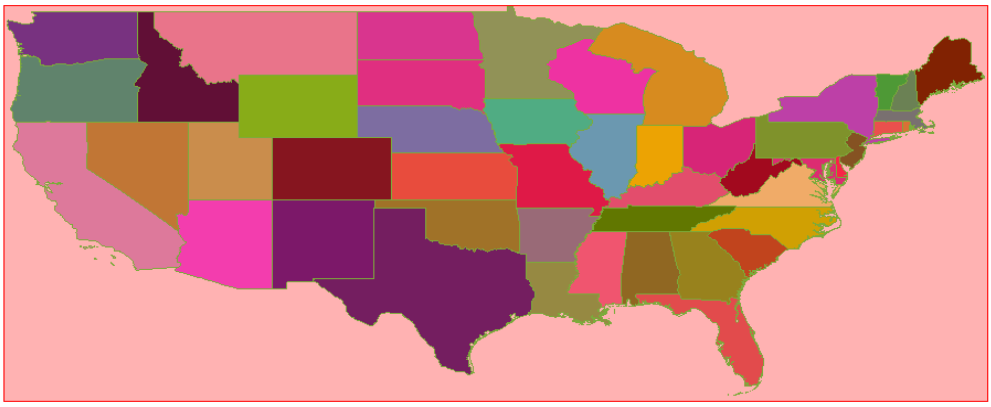
2.5. Data

The data chosen for the benchmark were:

- **2010 US Census Block Group Polygons**
  (231,423 polygons)
  http://www.gisnuts.com/terra/downloads2/Shapefiles/tl_2010_US_bg10.zip
- **US States**
  (48 polygons)
  http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/national_transportation_atlas_database/2013/zip/state.zip
- **OpenStreetMap** data:
  http://osmdata.thinkgeo.com/openstreetmap-data/north-america/us/
  - **Places, POIS & Railway Stations** stored in a single table
    (546,652 points)

- **Buildings**
  (3,091,498 polygons)
- **Roads**
  (15,624,811 lines)

- Multi-linestrings and multi-polygons normalised to single objects;
- Source data SRID (Spatial Reference System Identifier) is 4326;
- Alaska, Hawaii and Puerto Rico removed;
- Data is available for download as an Oracle EXP/IMP file and PostgreSQL export files;
- Data extent is:



2.5.1. Data Load

The Oracle Spatial and PostGIS datasets were subject to roughly the same load methods; shapefiles were loaded with standard tools:

- Oracle was loaded using SQL Developer's GeoRaptor's shapefile loader;
- PostGIS was loaded using "PostGIS 2.0 Shapefile and DBF Loader Exporter".

The tables that were created from the load were (the G_ prefix stands for Geodetic_ or long/lat 4326):

G_BLOCKGROUP_2010	G_BUILDING	G_PLACE
G_US_STATES	G_ROAD	

The spatial columns for each of these tables were named GEOG4326.

Once loaded, the G_* tables were then transformed into Google Mercator 3857. The resultant tables (with the M_ prefix for M[ercator]_) were creating using SQL scripts described in "Error: Reference source not found".

M_BLOCKGROUP_2010	M_BUILDING	M_PLACE
M_US_STATES	M_ROAD	

The spatial columns for these tables were named GEOMXXXX (where XXXX is the SRID value) e.g. GEOM3785.

The 3rd party spatial type's data was loaded directly from the Oracle SDO_GEOMETRY tables via the OGC's Well Known Binary (WKB) interchange format. The non-point SDO_GEOMETRY data was exported using the GET_WKB() SDO_GEOMETRY method and the passed to the 3rd Party Type's appropriate constructor:

```
TREAT(<<SCHEMA>>.ST_LINEFROMWKB(ora.GEOM3785.get_wkb(),3857)
   AS <<SCHEMA>>.ST_LINESTRING) as GEOM3857[1]
```

For point data the more direct XY ST_Point constructor was used instead of the WKB constructor.

1 The SRIDs are equivalent; it is just that one spatial type had a different SRID value for the same projection.

```
TREAT(<<SCHEMA>>.ST_POINT(ora.GEOM3785.sdo_point.x,
                          ora.GEOM3785.sdo_point.y,3857)
   AS <<SCHEMA>>.ST_POINT) as GEOM3857
```

The performance of the 3rd party spatial type constructors was so poor that Oracle's DBMS_PARALLEL_EXECUTE package was used to CHUNK the source data for writing into the target table (see Appendix B).

2.5.2. Validation and Geometry Type

The Oracle Spatial and PostGIS data were validated using the relevant validation tool after loading. For Oracle Spatial this is SDO_GEOM.VALIDATE_GEOMETRY and for PostGIS this is ST_isValid (with ST_MakeValid if any error occurred). The GIS 3rd Party Type's data was loaded from valid Oracle data via WKB.

For some datasets that were loaded, mainly polygon data, some multi-geometry (polygon and linestring) objects existed. These multi-geometry objects were very small in number: the dataset was therefore normalised to holding only polygon data before indexing and testing.

2.5.3. Timing Validation

The conducted tests all depend on a method of timing command of function execution that is similar. How this was determined for each database is included in page 33.

2.6. Geodetic and Cartesian/Planar Processing

Geodetic processing of spatial data is heavily computationally expensive; it requires complex calculations for such operations as area and length (a simple Pythagorean calculation for length is incorrect on a curved surface). Some object type implementations process geodetic/geographic coordinates using mathematics that treat latitude/longitude values as just Cartesian numbers c.f., Pythagoras.

A simple processing test was derived for geodetic objects to ensure that processing was indeed correct.

Oracle Database has a single type, SDO_GEOMETRY, and bases its processing decisions on the type of SRID. Specifically for this test it detects that 4326 is geodetic (longitude/latitude) and so bases all of its calculations and analysis on the appropriate mathematics; 3857 is treated as a planar projection so its calculations are Cartesian.

PostGIS separates processing by type: GEOMETRY for Cartesian /planar processing; GEOGRAPHY for geodetic processing (only SRID 4326). PostGIS still allows geographic/geodetic data to be stored in its GEOMETRY type but when stored in this way the processing is Cartesian.

The 3rd Party spatial data type is a single type named ST_GEOMETRY that stores both data types. It should also determine its processing based on the type of the SRID, however, testing (below) indicates that the type does not appear to implement geodetic processing.

2.6.1. Geodetic Processing

The following two tests demonstrate that both the Oracle Spatial and PostGIS primary filtering of geodetic data using a geodetic MBR (Minimum Bounding Rectangle) are correct.

Note that a geodetic MBR's sides are not "straight" as would be in a planar SRID. If they are treated as straight, a different search result will occur.

```
/* Oracle Spatial */
select count(*)
  from perf.g_road r
 where SDO_Filter(
r.geog4326,                  SDO_GEOMETRY(2003,4326,NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),                    SDO_ORDINATE_ARRAY(
  -77.279,38.745, -76.785,39.043))
  ) = 'TRUE';
--> 67421


/* PostGIS */
select count(*)
  from public.g_road r
 where r.geog4326
    && ST_MakeEnvelope(-77.279,38.745,-76.785,39.043,4326)::geography;
--> 67335
```

The difference between the two results is negligible.

The third spatial data type's processing of geodetic data is planar in nature as shown by the following query.

```
/* 3rd party spatial type */
select count(*)
  from gis.g_road r
 where SCHEMA.ST_xxxxIntersects[2](r.geog4326,
       SCHEMA.ST_ENVELOPE(SCHEMA.ST_MULTIPOINT(
                'MULTIPOINT(-77.279 38.745, -76.785 39.043)',4326)))
         = 1;
--> 61034
```

The proof that this result is based on Cartesian processing can be seen by forcing PostGIS to execute processing of the 4326 geographic data using the geometry object[3]:

```
select count(*) /* PostGIS: Cast to geometry to force Cartesian processing */
  from g_road r
 where r.geog4326::geometry
    && ST_MakeEnvelope(-77.279,38.745,-76.785,39.043,4326)::geometry;
--> 61038
```

The lack of geodetic processing in the 3rd party spatial type, means that comparisons requiring geodetic processing can only be performed between Oracle Spatial and PostGIS.

2.6.2. Table Creation and Sorting

All data tables are stored in Morton Key[4] spatial order (50km grid) with minimal spare space (e.g. PCTFREE 0 PCTUSED 99 in Oracle Spatial or FILLFACTOR=100 in PostgreSQL). Statistics were gathered for all created tables. The following example shows how the 4326 geodetic data is copied from G_BLOCKGROUP_2010 to 3785 M_BLOCKGROUP_2010 for Oracle SDO_GEOMETRY.

```
CREATE TABLE M_BLOCKGROUP_2010 (
  /* attributes from G_BLOCKGROUP_2010 removed */
  GEOM3785 MDSYS.SDO_GEOMETRY
) PCTFREE 0 PCTUSED 99 NOLOGGING;

INSERT /*+append */
  INTO PERF.M_BLOCKGROUP_2010
      (GID,/*Attributes Removed*/,morton_key,GEOM3785)
```

2 Actual operator schema.name obfuscated.

3 Storing the 4326 data in geometry(LineString,4326) object and querying returns the same result as the cast: in both cases the latitude/longitude data is processed as if it were Cartesian.

4 See http://spatialdbadvisor.com/oracle_spatial_tips_tricks/138/spatial-sorting-of-data-via-morton-key

```
WITH LL AS (
SELECT a.LL.sdo_point.x as x, a.LL.sdo_point.y as y, 50000 as gridSize
  FROM (SELECT SDO_CS.Transform(
                SDO_GEOMETRY(2001,4326,
                    SDO_POINT_TYPE(-124.755790710449,24.518321990967,NULL),
                    NULL,NULL),3785) LL
          FROM dual
       ) a
)
SELECT GID,/*Attributes Removed*/,morton_key,
       ST_RoundOrdinates(GEOM3785,3,3) as geom375
  FROM (SELECT GID,/* Attributes Removed*/,
               ST_Morton(FLOOR((a.point.sdo_point.y - (LL.Y))/LL.gridSize),
                         FLOOR((a.point.sdo_point.x - (LL.x))/LL.gridSize))
                 as morton_key,
               geom3785
          FROM LL,
               (SELECT GID,/* Attributes Removed*/,geom3785,
                       sdo_geom.sdo_pointonsurface(d.geom3785,0.005) as point
                  FROM (SELECT GID,/* Attributes Removed*/,
                               sdo_cs.transform (c.geog4326,3785) as geom3785
                          FROM PERF.G_BLOCKGROUP_2010 c
                        ) d
               ) a
        ) b
 ORDER BY morton_key;
COMMIT;

ALTER  TABLE M_BLOCKGROUP_2010 LOGGING;
ALTER  TABLE M_BLOCKGROUP_2010
         ADD CONSTRAINT M_BLOCKGROUP_2010_GID_PK PRIMARY KEY (GID) PCTFREE 0;
CREATE INDEX M_BLOCKGROUP_2010_county_idx
          ON M_BLOCKGROUP_2010(countyfp10) PCTFREE 0;
CREATE INDEX M_BLOCKGROUP_2010_state_idx
          ON M_BLOCKGROUP_2010(statefp10)  PCTFREE 0;

/* sdo_geom_metadata entry creation not shown*/
CREATE INDEX M_BGROUP_10_GEOM3785_SPX
          ON M_BLOCKGROUP_2010(geom3785)
   INDEXTYPE is mdsys.spatial_index
  PARAMETERS ('sdo_indx_dims=2, layer_gtype=POLYGON, sdo_rtr_pctfree=0')
PARALLEL 2;

EXEC DBMS_STATS.GATHER_TABLE_STATS (ownname => 'PERF',tabname =>
'M_BLOCKGROUP_2010',estimate_percent => 100);
```

2.7. Implementation Limitations

Each of the three implementations presented certain limitations that determined what could be compared.

2.7.1. Oracle Spatial

Some of the Oracle geo-processing functions are written close to the kernel in C or C++, however others are written in Java and hence run in the JVM. Such examples include:

- SDO_GEOM.SDO_BUFFER

  The SDO_BUFFER tests performed show that the Oracle Spatial Java implementation performs substantially slower than the PostGIS and 3rd Party implementations. While the relative difference of this function is significant compared to its competitors, the overall performance is no better than *usable* when executed individually. One should note that in practice this single processor performance difference can be mitigated substantially by running Buffer operations using Oracle Database parallel query.

- SDO_UTIL.FROM_WKBGEOMETRY / TO_WKBGEOMETRY
- SDO_UTIL.FROM_GMLGEOMETRY / TO_GMLGEOMETRY

- SDO_UTIL.FROM_GML311GEOMETRY / TO_GML311GEOMETRY
- SDO_UTIL.FROM_KMLGEOMETRY / TO_KMLGEOMETRY

The SDO_UTIL import/export functions are not the subject of this performance analysis. Though import/export routines such as TO_GMLGEOMETRY, speed is less of an issue, one still does not want large scale import/exports taking a long time. When used with other Oracle database technologies (such as DBMS_PARALLEL_EXECUTE) the duration of long running tasks, can be improved significantly.

2.7.2. 3rd Party Spatial Type

The 3rd party spatial type has the following difficulties:

- EXTPROC errors were common and difficult to correct;
- Constructors are very, very slow (a workaround was to parallel load the data using Oracle Database DBMS_PARALLEL_EXECUTE);
- Spatial index is not an RTree but is grid based:
  - Similar to Microsoft's Spatial index but doesn't have Microsoft's "auto" tuning support.
  - Trial and error methods for defining starting grid setting and optimising;
  - Geographic grids have to be declared in decimal degrees and not meters!
- No Geographic/Geodetic processing support;
  - ST_Buffer cannot take distance in meters[5] (Oracle Spatial and PostGIS can);
- No Nearest Neighbour implementation;
- No ability to determine topological (9Dim) relationship mask between two geometries (cf ST_Relate), unlike PostGIS (`text ST_Relate(geometry geomA, geometry geomB)`) or Oracle (SDO_GEOM.RELATE(geomA,'DETERMINE',geomB,0.005)).

2.7.3. PostGIS Data Types

2.7.3.1. GEOMETRY

The PostGIS GEOMETRY type is similar to the Oracle Database SDO_GEOMETRY object in that it can store geographic/geodetic data but, unlike Oracle, it does not use the SRID of the object to determine what processing to execute: thus, for geodetic SRIDs, it processes the data using planar arithmetic rather than using geodetic calculations. Where geodetic calculations are needed, the GEOGRAPHY data type must be used.

2.7.3.2. GEOGRAPHY

The GEOGRAPHY data type is recommended for where true geodetic calculations are needed. However, this data type has the following restrictions:

- Does **not** Support[6]:
  - SRIDs other than 4326
  - Secondary filters:

    ST_Contains ST_Crosses ST_Disjoint ST_Equals
    ST_Overlaps ST_Relate ST_Touches ST_Within
- Supports:
  - SRID 4326 only;
  - Secondary Filters:

5 Instead of 5 meters one would supply its equivalent (dependent on position on globe) in decimal degrees. For example, 0.0001 decimal degrees is roughly 11.1m at the equator, 10.3m at 23º N/S, 7.9m at 45º N/S, 4.3m at 67º N/S. These are all east/west distances. Having to compute a decimal degrees value for a distance in meters to supply to a function is unsatisfactory.

6 One can cast a geography object to geometry and use a secondary filter but processing will be Cartesian and therefore incorrect.

        - ST_CoveredBy
          (Not part of OGC standard)
        - ST_Covers
          (Not part of OGC standard: For geography only polygon covers point is supported)
        - ST_Intersects
    - Accessors such as ST_NumGeometries can only be called by casting the GEOGRAPHY object to a GEOMETRY object e.g.:

```
SELECT ST_NumGeometries(
          ST_GeogFromText('SRID=4326;POINT(153.029 -27.436)')::geometry);
```

2.8. Storage Comparison

It is important to understand that the three spatial types use different storage mechanisms and the effect this may have on comparison.

PostGIS appears to use an extended (e.g., includes SRID and Z/M ordinates) form of the Open Geospatial Consortium Well Known Binary (WKB) format; Oracle Spatial SDO_GEOMETRY is an object defined using the Oracle Database Object Type system. In particular, the main attribute of the SDO_GEOMETRY object for storing a geometry's ordinates is an array of NUMBER (SDO_ORDINATE_ARRAY); the 3rd Party Type also uses the Oracle Database Object Type system but stores its ordinates in a BLOB (whose internal format appears to be a form of WKB).

The 3rd Party Type appears to be a form of the table structure defined by the OGC's "7.1.5.3 Geometry stored using SQL binary types"[7]. This structure caches many properties of a geometry object: its number of points, its minimum bounding rectangle (MBR), its area and/or length. The MBR and number of points are stored in the header of each PostGIS geometry/geography object[8]. For Oracle Spatial all these properties are computed dynamically. While it is possible to persist this derived information with Oracle Spatial as well, this is not the default behaviour and is not the common practice. As such any comparison of methods that expose these values would be unfair and so are not attempted.

Each of the three spatial type APIs appear to be written in either C or C++ and use double precision floating point numbers to store each ordinate and for all computations. For the PostGIS and 3rd Party Type, there appears to be an alignment between the storage of a geometry's ordinates and the double precision representations used in computation: conversion overheads are therefore probably minimal. Oracle Spatial storage of ordinates is, as already pointed out, an array of NUMBER and not an array of binary double.

Oracle stores a NUMBER in a variable storage format so small NUMBERs take less space than larger NUMBERs[9]. This is a useful thing to know because it directly affects the size of a stored geometry object and thus can affect performance. A binary double's storage size is fixed at 8 bytes (64 bits) regardless as to the precision of the data being held: thus the size of a double precision ordinate is the same for a value of 1.01 and it is for 567900.348534.

One aspect of how Oracle stores ordinates in NUMBER format is that it requires conversion of the ordinates in the SDO_ORDINATE_ARRAY to in-memory double precision representation for computation. This imposes a conversion cost: the scale of that cost in comparison to the other two types

7 OpenGIS®Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option,Version 1.2.0, Page 34.
8 http://trac.osgeo.org/postgis/browser/trunk/liblwgeom/g_serialized.txt
9 See my article on the effect of rounding ordinate values at http://spatialdbadvisor.com/oracle_spatial_tips_tricks/208/saving-space-when-using-sdo_geometry-data

is difficult to quantify.

In short, coordinate values are stored in two ways by the spatial type systems being examined:

- Fixed size, floating point numbers based on the IEEE Floating-Point Standard (double precision)
- Variable sized SQL numeric type values;

2.9. Precision Model

Floating point numbers do not, of themselves, have any sort of metadata that describes their numeric precision. Even though a number was observed to 2 decimal places of precision, there is no internal property that records that precision to allow software to take into account the precision when computing new values with other numbers with difference precision. For example, if we multiplied 12.01 (2 decimal digits of precision) x 12.1 (1 decimal digit) then the number could only be 145.3 and not 145.321.

So we can see that some sort of statement of precision, **a precision model**, is required when processing spatial ordinates to ensure that answers respect their inherent precision. For example, to determine if two adjacent coordinates are the same when checking the validity of the description of a spatial object, some sort of precision statement is required. When, for example, are the following two coordinates equal, at the *centimetre*, *millimetre*, *micrometre*, or some other decimal place?

```
515343.140982008,5216871.7926608
515343.141052,   5216871.79316
```

A precision model is the metadata that describes how two coordinates are to be compared when determining equality or when calculating a new coordinate value from two coordinates with different ordinate precision. Generally, coordinate ordinate precision is a statement of the scale of a number; for example, a scale greater than one means that the precision point is to the right of the decimal point.

Only Oracle Spatial has a form of a precision model: the other two evaluated do not. This model is exposed through the metadata it stores and the use of that metadata in functions.

2.9.1.1. Oracle Spatial XXX_SDO_GEOM_METADATA

Oracle Spatial requires certain metadata before the spatial data can be meaningfully used by applications. There are two basic database views defined to store this metadata information: USER_SDO_GEOM_METADATA and ALL_SDO_GEOM_METADATA. The views are set up so that owners of the spatial tables or views can create the metadata for them. The views are used by Oracle Spatial to associate SDO_GEOMETRY with metadata that defines a number of items including the SRID and tolerance values for each of four dimensions:

```
TYPE SDO_DIM_ARRAY AS VARRAY(4) OF SDO_DIM_ELEMENT

TYPE SDO_DIM_ELEMENT AS OBJECT(
   SDO_DIMNAME     VARCHAR(64),     -- X, Y, Z, M, LATITUDE, LONGITUDE etc
   SDO_LB          NUMBER,          -- Lowest value in ordinate range
   SDO_UB          NUMBER,          -- Highest value in ordinate range
   SDO_TOLERANCE   NUMBER)          -- tolerance value for dimension.
```

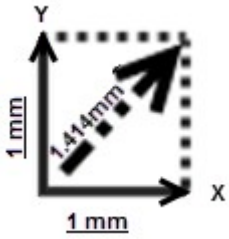
The structure is stored in the DIMINFO column of the USER_SDO_GEOM_METADATA table:

```
INSERT INTO user_sdo_geom_metadata (table_name, column_name, diminfo, srid)
VALUES('M_BLOCKGROUP_2010',
       'GEOM3785',
       SDO_DIM_ARRAY(
         SDO_DIM_ELEMENT('X',-13887750.0,-7453285.5,0.005),
         SDO_DIM_ELEMENT('Y',  2816690.0, 6340548.2,0.005)),
       3785);
COMMIT;
```

The SDO_TOLERANCE value of **0.005** for X and Y represents a precision of 1cm. This value allows Oracle Spatial to determine the level of precision when *comparing* or *processing* coordinates: it does not enforce this tolerance value by rounding the ordinate values of an SDO_GEOMETRY column to a specific precision in response to Data Manipulation Language (DML) actions (INSERT or UPDATE). Other than its use in processing and comparison, the main reason for understanding how this SDO_TOLERANCE value is applied is because an SDO_TOLERANCE value is:

> *[the] distance that two points can be apart and still be considered the same (see "1 Spatial Concepts, 1.5.5 Tolerance " in Oracle® Spatial Developer's Guide, 11g Release 2 (11.2), Part Number E11830-04).*

Put another way, this is a distance between two coordinates, which is not the same as the precision of a single ordinate's decimal number scale value. Thus, applying a tolerance to a stored ordinate would be incorrect.



2.9.1.2. Functions and Metadata

The precision model used by Oracle Spatial is exposed through its geo-processing functions. The following example of SDO_GEOM shows how Oracle Spatial functions can get its metadata from a DIMINFO structure or a simple tolerance value.

```
FUNCTION SDO_GEOM.SDO_UNION(GEOM1 IN SDO_GEOMETRY,
                            DIM1  IN SDO_DIM_ARRAY,
                            GEOM2 IN SDO_GEOMETRY,
                            DIM2  IN SDO_DIM_ARRAY)
  RETURN SDO_GEOMETRY;

FUNCTION SDO_GEOM.SDO_UNION(GEOM1 IN SDO_GEOMETRY,
                            GEOM2 IN SDO_GEOMETRY,
                            TOL   IN NUMBER)
  RETURN SDO_GEOMETRY;

FUNCTION SDO_GEOM.VALIDATE_GEOMETRY_WITH_CONTEXT(
                        theGeometry IN SDO_GEOMETRY,
                        tolerance   IN NUMBER,
                    [or theDimInfo  IN SDO_DIM_ARRAY, ]
                        conditional IN VARCHAR2 DEFAULT 'TRUE' ,
                        flag10g     IN VARCHAR2 DEFAULT 'FALSE')
  RETURN VARCHAR2;
```

The equivalent functions for PostGIS and the 3rd Party Type do not allow the input of variable precision metadata.

```
Geometry ST_Union(geometry g1, geometry g2);
text     ST_IsValidReason(geometry geomA);
```

It is not any one product's approach is right, but it is something that must be kept in mind when comparing products.

2.9.1.3. Applying Precision to Round Ordinates: Situations

Here are a few situations where the application of precision to an ordinate value may be justified.

1. When loading data from an external source, such as an ESRI shapefile, there is a mismatch between the way a shapefile stores ordinates in double precision format and the Oracle Spatial NUMBER-based storage.

The following SDO_GEOMETRY object, loaded from a shapefile, shows the mismatch between storage types. (The 28355 SRID is for "GDA94 / MGA zone 55" and the data is from 1:25,000 scale data, that is, +/- 15m on the ground.)

```
SDO_GEOMETRY(2003,28355,NULL,
             SDO_ELEM_INFO_ARRAY(1,1003,1),
             SDO_ORDINATE_ARRAY(
                 515343.140982008,5216871.7926608,
                 515358.672727787,5216871.7926608,
                 515344.832,5216879.5,
                 515343.140982008,5216871.7926608))
```

2. When processing two SDO_GEOMETRY objects with differing ordinate precision against each other (for example, SDO_DIFFERENCE and SDO_UNION). For example, compute the difference between a polygon and a linestring geometry where the polygon has lower precision than the linestring. Also, note that we gave a tolerance value of 0.05 (5cm) to the function for its processing.

```
SELECT SDO_GEOM.SDO_DIFFERENCE(
                     SDO_GEOMETRY(2002,28355,NULL,
                         SDO_ELEM_INFO_ARRAY(1,2,1),
                         SDO_ORDINATE_ARRAY(515350.171,5216870.41,
                         515343.435,5216876.411,515352.113,5216878.676)),
                     SDO_GEOMETRY(2003,28355,NULL,
                         SDO_ELEM_INFO_ARRAY(1,1003,1),
                         SDO_ORDINATE_ARRAY(515343.1,5216871.8,
                         515358.7,5216871.8,515344.8,5216879.5,
                         515343.1,5216871.8)),
                     0.05) as geom
   FROM dual;

-- Results
--
GEOM
------------------------------------------------------------
SDO_GEOMETRY(2006,28355,NULL,
             SDO_ELEM_INFO_ARRAY(1,2,1,7,2,1,11,2,1),
             SDO_ORDINATE_ARRAY(515344.159777912,5216876.60017054,
                                515343.435,5216876.411,
                                515344.005752361,5216875.9025254,
                                515348.610753375,5216871.8,
                                515350.171,5216870.41,
                                515348.153198659,5216877.64247269,
                                515352.113,5216878.676))
```

3. The projection or transformation of an SDO_GEOMETRY object from one coordinate system to another will change the number of decimal digits in the resulting ordinates. Let's assume we have a longitude/latitude value (generated via a click on a map) and we want the Google Mercator Map coordinates for that point.

```
SELECT SDO_CS.Transform(
              SDO_GEOMETRY(2002,8311,NULL,
                       SDO_ELEM_INFO_ARRAY(1,2,1),
                       SDO_ORDINATE_ARRAY(147.123,-32.456,147.672,-33.739)),
              3785) as geom
  FROM dual;

-- Results
-
GEOM
```

```
----------------------------------------------------------
SDO_GEOMETRY(2002,3785,NULL,
             SDO_ELEM_INFO_ARRAY(1,2,1),
             SDO_ORDINATE_ARRAY(16377657.4439788,-3800381.82007675,
                                16438771.8444243,-3970070.49100647))
```

Being that the input data was only specified to 0.001 of a degree, an output—in meters—specified to eight decimal places seems somewhat excessive!

What constitutes a suitable precision value depends on the original "sensor" that recorded the original value:

1. Manually surveyed (theodolite and surveyor) may be both accurate and precise, recording observations down to 1 mm.
2. High precision differential GPS may record ordinate values down to 1 cm.
3. Cheap hand-held GPS may record a specific coordinate value to a few meters or tens of meters of accuracy.
4. Data scanned from old paper/Mylar maps may be only accurate to ±10 to ±20 meters!
5. Satellite data is variable in accuracy and precision.

3. Tests

3.1. Flipping the Switch

The default settings in Oracle Database 12c do not enable an important new Oracle Spatial 12c performance feature. To activate these performance improvements requires Oracle Database Enterprise Edition with an Oracle Spatial 12c license; a new parameter called SPATIAL_VECTOR_ACCELERATION must be set to true. This parameter can be set for the whole system or for a single session. For the whole system either enter the following statement from a suitably privileged account:

```
ALTER SYSTEM SET SPATIAL_VECTOR_ACCELERATION = TRUE;
```

Or add the following to the database initialisation file (xxxinit.ora):

```
SPATIAL_VECTOR_ACCELERATION = TRUE;
```

If you wish to set the value for the current session, enter the following from a suitably privileged account:

```
ALTER SESSION SET SPATIAL_VECTOR_ACCELERATION = TRUE;
```

3.2. Scope of Tests

While a larger set of tests were considered, the relative capabilities of the comparison data types reduced the set to the following:

- Primary Filtering;
- Secondary Filtering with simple masks (ANYINTERACT, INSIDE, CONTAINS)
- Aggregation;
- Distance computations;
- Nearest neighbour.

The following tests of functionality and performance were not attempted.

- Union/Intersection operations between two objects;
- Data loading and index creation;
- Queries with multiple predicates (one spatial and one or more non-spatial);
- Spatial joins between two or more tables.

The following test was attempted but limitations in the non-Oracle products made fair comparison difficult.

- Relation operations (i.e. determining spatial relationship between two objects);

3.3. Primary Filtering

All spatial searching commences with the primary filtering of the data to return a candidate set of objects for which additional, more computationally expensive, processing can be enacted. Primary filtering is always done using the type's spatial index.

The approach to primary filtering involved creating a procedure that accepts:

- Number of tests to conduct (1000);
- Schema, Table and spatial Column to query (PLACE point, ROAD line and BLOCKGROUP_2010 polygon);
- A list of search windows (square) expressed in meters e.g., 5000,15000,25000,50000,100000 meters.

No geography MBR testing was done using the 3rd Party Type.

3.3.1. Planar Results

The following table totals all planar results for all object types, tables and search windows across 1,000 searches.

Geometry Object Type	Table (Geometry Type)	Time (s) / Search	Features / Search	Features / Second	Faster (x) than slowest
PostGIS	M_BLOCKGROUP_2010 (Polygon)	0.00205	238.6	116,674	3.73
Oracle Spatial		0.00248	236.0	95,212	3.0
3rd Party Type		0.00723	225.8	31,251	1.0
Comments: The difference between the three types is negligible in terms of average time per search; Oracle Spatial and PostGIS were fastest in terms of features/second, with PostGIS being slightly faster than Oracle Spatial.					
PostGIS	M_ROAD (Linestring)	0.01202	15,507.5	1,290,585	23.8
Oracle Spatial		0.01960	14,456.6	737,505	13.6
3rd Party Type		0.25300	13,746.4	54,332	1.0
Comments: PostGIS is two times faster than Oracle Spatial in terms of features per second yet there is little between them in terms of seconds / search.. The 3rd Party Type is an order of magnitude slower than both.					
PostGIS	M_PLACE (Point)	0.00178	532.930	300,126	5.3
Oracle Spatial		0.00364	524.872	144,387	2.5
3rd Party Type		0.00928	529.719	57,081	1.0
Comments: The table shows that the difference between PostGIS and Oracle Spatial in terms of average search time is negligible with the 3rd Party Type still delivering reasonable per search performance. PostGIS's throughput is again better than its competitors.					

3.3.2. Geodetic Results

The following table totals all geodetic results for all object types, tables and search windows across 1000 searches.

Geometry Object Type	Table (Geometry Type)	Time (s) / Search	Features / Search	Features / Second	Faster (x) than slowest
PostGIS	G_BLOCKGROUP_2010 (Polygon)	0.00253	72.9	28,784	1.1
Oracle Spatial		0.00270	386.2	142,782	1.0
Comments:	The difference between Oracle Spatial and PostGIS is negligible if one considers the time per search, but when one takes into account throughput (features per second) Oracle Spatial delivers much better performance.				
PostGIS	G_ROAD (Linestring)	0.00703	3,649.1	519,111	3.6
Oracle Spatial		0.02534	27,016.6	524,230	1.0
Comments:	The table shows that the difference between Oracle Spatial and PostGIS is negligible in terms of throughout, though PostGIS was 3.5 times faster in terms of average seconds per search.				
PostGIS	G_PLACE (Point)	0.00249	144.417	58,002	1.8
Oracle Spatial		0.00417	912.353	219,029	1.0
Comments:	While PostGIS average search time is faster, when normalised by features processed, Oracle Spatial performance is much better.				

3.4. Secondary Filtering

Secondary filtering involves the accurate examination of the coordinates of primary filtered objects to determine topological properties such as inside, touch etc. Only Oracle Spatial and PostGIS have the CoveredBy and Covers secondary operators; as such these are not reported.

Most of the geometry type pairs (polygon->polygon; polygon->line) were attempted but point within polygon (State, BlockGroup and Building) was not attempted due to a lack of time[10].

3.4.1. Planar Results

Geometry Object Type	Secondary Filter	FILTER TABLE (Geometry Type) SEARCH TABLE (Geometry Type)	Average Search Time (s)	Total Feats	Feats / Sec	Faster (x) than Slowest (1)
Oracle Spatial	SDO_Touch	M_BLOCKGROUP_2010 (Polygon)	0.0046	616	1,343	7.9
3rd Party Type	ST_Touches		0.0103	650	634	3.7
PostGIS	ST_Touches	M_BLOCKGROUP_2010 (Polygon)	0.0375	640	171	1
Comments:	All three types returned similar total number of features (all within 3% of average), yet normalised by number of features per second, the differences become more significant: Oracle is nearly an order of magnitude faster than PostGIS and is nearly twice as fast as the 3rd Party Type. All types appear to provide sufficient performance for normal, interactive use though Oracle Spatial results would provide better scalability under load.					
Oracle Spatial	SDO_AnyInteract	M_BLOCKGROUP_2010 (Polygon)	0.0089	7,997	9,024	11.4
PostGIS	ST_Intersects		0.0157	8,577	5,482	6.9
3rd Party Type	ST_Intersects	M_ROAD (Linestring)	0.0993	7,886	794	1
Comments:	AnyInteract or Intersects is a coarse secondary filter that looks for any type of interaction and not a specific one (eg inside). It is normally the fastest secondary filter available for each type. For polygon-intersects-line the numbers mainly endorse this assertion. Oracle Spatial is 10 times faster than the slowest; and twice as fast as PostGIS.					
Oracle Spatial	SDO_Inside	M_BLOCKGROUP_2010 (Polygon)	0.0033	4,634	14,139	21.3
PostGIS	ST_Within		0.0242	5,358	2,211	3.3

10 This was a major oversight of the testing.

Geometry Object Type	Secondary Filter	FILTER TABLE (Geometry Type) SEARCH TABLE (Geometry Type)	Average Search Time (s)	Total Feats	Feats / Sec	Faster (x) than Slowest (1)
3rd Party Type	ST_Within	M_ROAD (Linestring)	0.0871	5,788	664	1
Comments:	Finding lines inside polygons involves significant computations. The average times for Oracle Spatial and PostGIS demonstrate a speed that is probably sufficient for most real-world requirements. The 3rd Party Type while significantly slower produced an average speed that indicates that the secondary operator is also fit for use in most real world situations.					
Oracle Spatial	SDO_AnyInteract	M_US_STATES (Polygon)	0.0969	484,120	49,976	41.6
3rd Party Type	ST_Intersects		0.4777	368,170	7,707	6.4
PostGIS	ST_Intersects	M_BLOCKGROUP_2010 (Polygon)	3.8944	467,424	1,200	1
Comments:	Polygon-on-polygon operations always involve significant processing. The large nature of the state polygons and complete coverage within them of small block groups shows why so many features were processed. While intersection is a coarse secondary filter, the results for Oracle Spatial are especially meritorious and fit for purpose in most applications. The 3rd Party Type's performance, while significantly slower still demonstrated reasonable performance given the nature of the data.					
Oracle Spatial	SDO_Inside	M_US_STATES (Polygon)	0.1968	461,359	23,442	34.6
3rd Party Type	ST_Within		0.5517	500,298	9,068	13.4
PostGIS	ST_Within	M_BLOCKGROUP_2010 (Polygon)	5.5518	375,896	677	1
Comments:	Again the inside statistics show that the computation is more significant that simple intersection. The previous comments on the results apply to this test.					
Oracle Spatial	SDO_Inside	M_US_STATES (Polygon)	1.4191	35,941,383	253,277	54.0
3rd Party Type	ST_Within		52.6201	29,718,083	5,648	1.2
PostGIS	ST_Within	M_ROAD (Linestring)	70.7204	33,189,831	4,693	1
Comments:	The large volume of data returned is the reason for the > 1second results. Oracle Spatial performance is particularly noteworthy. For PostGIS and the 3rd Party Type, the performance while ostensibly much worse that for Oracle Spatial should be taken within the context of the large number of lines processed: such an operation would probably not normally be done interactively.					

Overall a consistent set of results with Oracle appearing to provide the best results for all secondary filters across all pairs of geometry types.

3.4.2. Geodetic Results

As pointed out above, the PostGIS secondary operators that support geographic data:

- ST_Covers Only Polygon covers point is supported
- ST_CoveredBy Points, lines and polygons
- ST_Intersects Points, lines and polygons

Only the ST_Intersects / SDO_AnyInteract methods were tested.

Geometry Object Type	Secondary Filter	Table (Geometry Type)	Time (s) / Search	Feats / Search	Feats / Sec	Faster (x) than Slowest (1)
Oracle Spatial	SDO_AnyInteract	G_BLOCKGROUP_2010 (Polygon)	0.018	9	5,063	31.4
PostGIS	ST_Intersects	G_ROAD (Linestring)	0.592	10	161	1
Oracle Spatial	SDO_AnyInteract		0.510	417.35	8,181	247.9

Geometry Object Type	Secondary Filter	Table (Geometry Type)	Time (s) / Search	Feats / Search	Feats / Sec	Faster (x) than Slowest (1)
PostGIS	ST_Intersects	G_US_STATES (Polygon) G_BLOCKGROUP_2010 (Polygon)	120.084	397.71	33	1
Oracle Spatial	SDO_AnyInteract	G_US_STATES (Polygon)	4.662	33,814.3	72,539	348.7
PostGIS	ST_Intersects	G_ROAD (Linestring)	1,378.515	3,734.2	208	1

These tests show that Oracle Spatial performance for processing geodetic data is stunningly good, though some balance may have been presented if its other secondary geodetic operators had been tested. The performance of PostGIS generic ST_Intersects geodetic operator, given its ability with its planar equivalents, is surprisingly poor. PostGIS GEOGRAPHY type and operators are at an early stage in development whereas Oracle Spatial geodetic support has been in existence for over 15 years.

3.5. Distance

The distance test determined the average time taken to calculate the distance between two objects. The objects were drawn from two tables. The first table is one of the base tables, the second, to reduce the number of tests, was created from a sub-set of objects drawn from the ROAD table e.g. M_SMALL_ROAD (exported/imported into all spatial type databases). The comparison SQL is as follows:

```
SELECT SUM(ST_DISTANCE(a.geom3875,b.geom3875)), count(*)
  FROM M_BLOCKGROUP_2010 a,
       M_SMALL_ROAD      b
 WHERE a.gid BETWEEN $1 AND $2;
```

The GID values that limit the **outer join** are computed as follows.

1. The minimum and maximum GID values and range is computed from the actual data of table 1 (a).

```
SELECT max(gid)-min(gid) as gid_range,
       min(gid)          as mingid,
       max(gid)          as maxgid
  FROM M_BLOCKGROUP_2010 a;
```

2. For each loop (user specified e.g. 100) a starting GID is randomly generated in the range minimum to maximum. The finishing GID, is computed by adding 5 to the minimum GID (GIDs for a table are, or should be, sequential).

```
v_sql := ' SELECT SUM(ST_DISTANCE(a.'||v_column||',b.'||v_column||
                    ')), count(*) ' ||
           ' FROM public.' || p_dist_table1 ||' as a,' ||
                ' public.' || p_dist_table2 ||' as b '  ||
          ' WHERE a.gid between $1 and $2';
v_i := 1;
While (v_i <= v_loops) Loop
  begin
    v_gid         := (v_min_gid + (random() * v_gid_range))::integer;
    v_out_count   := -1;
    v_Start_Time  := extract(epoch from cast(timeofDay() as timestamp(6)))::Numeric;
    EXECUTE v_sql
       INTO v_sum, v_out_count
      USING v_gid, v_gid + 5;
    v_End_Time    := extract(epoch from cast(timeofDay() as timestamp(6)))::Numeric;
    v_seconds     := v_End_Time - v_Start_Time;
    RAISE INFO '%,%,%,%',
```

```
                v_gid,
                v_sum,
                v_out_count,
                trim(TO_CHAR(v_seconds,'FM999990.9999990'));
    v_i := v_i + 1;
    Exception
       WHEN ….....
  end;
End Loop;
```

The table below shows the average time taken across all loops. The total distance per distance check is assumed to not affect the average time or is assumed to have minimal effect.

Spatial Type	TABLE1 (GeometryType) TABLE2 (GeometryType)	Average Time	Faster (x) than Slowest (1)
PostGIS	M_BLOCKGROUP_2010 (Polygon) M_SMALL_ROAD (LineString)	0.037	17
Oracle Spatial		0.573	1.1
3rd Party		0.610	1.0
Comments:	PostGIS is an order of magnitude faster than its competitors. With Oracle Spatial the average of half a second per search is surprisingly slow and appears marginal for dynamic on-line querying, but this depends on the specific use. Certainly its performance could be improved. Oracle Spatial has a tolerance model so improvements could occur with more coarse parameters.		
3rd Party	M_BLOCKGROUP_2010 (Polygon) M_US_STATES (Polygon)	0.537	3.9
SDO		2.009	1.0
PostGIS		2.098	1
Comments:	The results here are the opposite of the previous. While the speed difference is close, the actual average speed difference would be significant for where live operator use. Both tables contain complicated polygons: this may be a factor in the slowdown compared to the first.		
PostGIS	**M_ROAD (LineString) M_US_STATES (Polygon)**	0.248	4
3rd Party		0.457	2.3
Oracle Spatial		1.065	1.0
Comments:	This is a similar test to the first in that the tables contain polygons and linestrings. The road objects are not complex, while the states ones are. This may account for the slower speed of this test compared to the first.		

3.6. Aggregation

The aggregation operators are not something covered by the relevant OGC or SQL/MM standard. Only the **union** operator is considered in this comparison. The union aggregator for each type is as follows:

Type	Support
3rd Party Type	This spatial type has an aggregate union operator. However, as noted, this type does not appear to support processing of geographic/geodetic data other than Cartesianally (treat long/lat as X/Y).
PostGIS	ST_Union operator acts as an aggregator when acting on a geometry set. ST_Union only supports Cartesian data.
Oracle Spatial	The Oracle Spatial aggregator supports both Cartesian and geodetic data.

NOTE: The lack of support of an aggregate union operator for geodetic data is probably not a problem where one is trying to create a higher set of polygons from lower level set of nested polygons (e.g., a natural hierarchy of objects from Block Groups up to States as in the test data); if these data share a unique set of nodes/vertices and no more need to be computed, then such aggregates can be used to process geodetic data. Where the data being unioned contains overlapping polygons that do not share nodes, geometric processing is required.

The 3rd party type could not produce stable results for all tests conducted (only the states with FIPS codes 10, 31, 33, 44 and 50 – 5/48 or 10% - succeeded).

PostGIS's aggregate union operator does not support geographic/geodetic aggregation (Oracle does).

The Oracle Spatial type supports aggregation of geographic/geodetic and Cartesian data.

As such the only comparison executed was for Oracle Spatial (SDO_AGGR_UNION) vs PostGIS (ST_Union) for the Google Mercator data.

The aggregate union was conducted against the US Census 2010 block group data by aggregating the data based on the STATEFIPS code (48 states). For each the number of vertices in the final object was computed (for comparison purposes).

For example:

Oracle Spatial

```
SELECT CASE WHEN b.ugeog4326 IS NULL
            THEN -1
            ELSE sdo_util.GetNumVertices(ugeog4326)
       END as result
  INTO v_result
  FROM (SELECT Sdo_Aggr_Union(SdoAggrType(a.geog4326,0.05)) as ugeog4326
          FROM G_BLOCKGROUP_2010 a
         WHERE a.statefp10 = rec.statefp10
           AND a.geog4326 IS NOT NULL
         GROUP BY a.STATEFP10
        ) b;
```

PostGIS

```
SELECT CASE WHEN b.uGeog4326 IS NULL
            THEN -1
            ELSE ST_NPoints(b.uGeog4326::geometry)
       END as result
  INTO v_result
  FROM (SELECT ST_Union(a.geog4326::geometry)::geography as uGeog4326
          FROM public.g_blockgroup_2010 a
         WHERE a.statefp10 = v_rec.STATEFP10
           AND a.geog4326 IS NOT NULL
         GROUP BY a.STATEFP10
       ) as b;
```

3.6.1. Projected Aggregation

With Oracle, 3 states failed to aggregate throwing the internal error ORA-13050 (unable to construct spatial object). All comparisons excluded the failed objects.

Type	Total (s) (100 Tests)	Avg (s) (100 Tests)	Faster (x) than Slowest (1)
Oracle Spatial	434.939	9.665	8.6
PostGIS	3,729.626	82.881	1

Discussion of results is after Geodetic Aggregation results.

3.6.2. Geodetic Aggregation

With Oracle, 9 states failed to aggregate throwing the internal error ORA-13050 (unable to construct spatial object). All comparisons excluded the failed objects.

Type	Total (s) (100 Tests)	Avg (s) (100 Tests)	Faster (x) than Slowest (1)
Oracle Spatial	481.549	12.347	5.5
PostGIS	2,655.105	68.080	1

3.6.3. Aggregation Discussion

Prior to Oracle Spatial 12c with spatial vector acceleration, the SDO_AGGR_UNION operator had been a poor performer for many, many years. While PostGIS's ST_Union aggregation implementation has been an excellent performer since PostGIS version 1.4[11] it is surprising to see how much faster Oracle Spatial was for this test.

The number of vertices returned in the constructed polygon object varied. The same number of vertices were returned in 47% of the time, with PostGIS returning smaller objects (less vertices) than Oracle. The vertex range was from 24% (eg for FPIS code 53 PostGIS returned an object with 7158 vertices, while Oracle returned one with 29,958. The average across all results was 93%. Oracle Spatial uses a tolerance which may have been the reason for the difference in results: the value used for SDO_AGGR_UNION was 0.05m or 5cm.

Even given this, the difference in relative performance (5.5 and 8.6) is not great. Both products appear to perform at acceptable levels given the particular activity (on line or transactional applications would normally not execute aggregation operations for interactive activities in which humans would be forced to wait for the result).

3.7. Nearest Neighbour

Nearest neighbour functionality is the ability of the spatial type to discover the nearest set of objects to a given object. That set can be:

1. The spatially nearest N neighbouring objects by distance;
2. The nearest N neighbouring objects by distance and filtered by some predicate eg nearest Bed and Breakfast businesses from a set of all accommodation objects.

While both these operations are implemented mainly via spatial index processing, the latter includes implementing attribute predicates with the spatial processing in an optimal manner (as determined via a wider application of the database's query optimiser).

The nearest neighbour tests in this analysis implements the first of the two above nearest neighbour search types. The support for NN processing is variable across the three data types:

Type	Support
3rd Party Type	***Does not have*** nearest neighbour functionality.
PostGIS	Supports NN for Cartesian data only but not for geographic/geodetic data; returning the actual distance with the candidate record must be done via an independent call to the ST_Distance function. It is important to note that the NN operator for PostGIS

11 See http://blog.cleverelephant.ca/2009/01/must-faster-unions-in-postgis-14.html

	"<->" : "Returns the distance between two points. For point / point checks it uses floating point accuracy (as opposed to the double precision accuracy of the underlying point geometry). For other geometry types the distance between the floating point bounding box centroids is returned. Useful for doing distance ordering and nearest neighbor limits using KNN gist functionality."[12]
Oracle Spatial	Supports NN processing for both geodetic and Cartesian data; the NN processing includes an ancillary operator (SDO_NN_DISTANCE) that returns the distance associated with the returned object (independent computation is not required).

Due to this support matrix, the only comparison executed was for Oracle SDO vs PostGIS for the Google Mercator data (Cartesian). An example of the SQL for this is as follows:

Oracle Spatial

```
SELECT avg(sdo_nn_distance(1))
  INTO v_avg_dist
  FROM perf.g_place a
 WHERE SDO_NN(a.geog4326,v_geom,'sdo_num_res='||v_res,1) = 'TRUE';
```

PostGIS

```
WITH index_query AS (
   SELECT ST_Distance(geom, v_geog::geometry) as nn_distance
     FROM public.g_place_a
    ORDER BY geom <-> v_geog::geometry
    LIMIT (v_res*2)
)
SELECT avg(nn_distance)
  INTO v_avg_dist
  FROM index_query
 ORDER BY d limit v_res;
```

The PostGIS SQL implements a method that the author guesses will return the correct result. Since the query is against point data (G_PLACE), this is probably correct. If the data being queried contained linear or polygon data, computation using surrogate centroids will result in an incorrect answer if LIMIT (v_res) was used. The author chose to use LIMIT(v_res*2) but this is dependent on the results being returned in actual distance order AND that the line/polygon centroids are accurate surrogates for their geometries. It is impossible to prove the latter so the correctness of the result is dependent on the actual nearest neighbour geometries being within the number returned by <-> and LIMIT.

Type	Avg (seconds) (100 Tests)	Avg Number Results / Query	Avg Distance (m) (Single Result)	Faster (x) than Slowest (1)
PostGIS	0.0022	259.89	7,625.3	11.8
Oracle Spatial	0.0260	234.67	34,827.5	1

3.7.1. Discussion

From the 100 tests, the average number of seconds to return over 200 records is 10 times faster for PostGIS than for Oracle.

However, remember we are not **strictly** comparing apples-with-apples at the individual function level: Oracle does all the work in the actual SDO_NN operator **calculates the actual minimum distance between the geometries.** SDO_NN_DISTANCE simply exposes the calculated distance value. PostGIS's operators (<-> and <#>) only ever return an approximate distance (for <-> this is the distance between the bounding box centroids – however the centroid here is the same as the G_PLACE point). To calculate a true distance, a call to ST_Distance must then be made (see reference SQL). One must remember to construct such SQL and to get the LIMIT clauses correct especially for non-point data.

12 http://postgis.net/docs/manual-2.1/geometry_distance_centroid.html

PostGIS performance is still impressive. PostGIS support for only point-to-point comparisons in its NN operators introduces SQL complexity and a measure of guesswork in its use. Given this, the Oracle Spatial single function implementation that supports all geometry types takes on a new light.

The actual speed taken for both spatial types against post data means that both can be used in interactive queries (though the PostGIS implementation may scale slightly better than Oracle Spatial). Oracle is preferred given its full implementation and simplicity of use.

4. Conclusion

In the spatial database world performance is a vitally important benchmark for the delivery of solutions to business requirements but even if performance is less than a competitor, the more critical question is:

"***Is the available speed, and functionality, fit for purpose***"?

In other words, will it allow business functions to be implemented quickly and perform in a mix of situations? For many Oracle Spatial practitioners functionality such as **SDO_AGGR_UNION** have not been fit for purpose for many applications. Having recourse to the well known "Appendix D" **"group by mod(rownum,16)"** workaround, or the later added **SDO_AGGR_SET_UNION**, helped, but they could only improve performance so far. Overall the fitness of the pre 12c aggregate union offerings were still were not fast and simple enough to use.

But this is now no longer the case. This analysis revealed an enormous performance improvement in *all the Oracle Spatial 12c functions tested with spatial vector acceleration invoked*. Simplicity of use, and functional coverage are also just as important as straight performance. So it is pleasing that this analysis has revealed that the Oracle Spatial functional offerings and simplicity of use for geodetic data are superior to the analysed competitors.

It is now true to say that Oracle Spatial 12c, on non-engineered systems, is equal to, or better than, its competitors (open source or commercial).

Appendix A: Buffer

Buffering is the expansion (+ve buffer values) or contraction (-ve buffer values) of a geometric object.

Type	Support
3rd party (using EXTPROC)	This spatial type does not appear to support processing of geographic/geodetic data other than Cartesianally (treat long/lat as X/Y). The buffer distance for geographic/geodetic data has to be specified in decimal degrees. Given that a decimal degree of latitude varied with latitude and is different to a decimal degree of longitude, what does a distance specified in decimal degrees mean? Because of this, the other two types (and SQL Server) all require the buffer distance to be specified in meters for geographic/geodetic data. Because of this perceived lack of support for correct processing of such data, only buffering of the Google Mercator data was attempted.
PostGIS	The ST_Buffer function supports both geodetic/geographic and Cartesian data. The geography buffer distance is specified in meters.
Oracle Spatial	The SDO_Buffer function supports both geodetic/geographic and Cartesian data. The geography buffer distance is specified in meters. However, as already noted, the Oracle implementation is in Java. The comparisons are provided here as an appendix.

The approach to testing buffering was as follows. The 2010 Block Group polygons primary keys were processed to generate a range from smallest to largest (load processing should ensure that no gaps exist in the range); then a PL/SQL (or pl/pgSQL) function/procedure block was written that would generate 100 GID values randomly from the defined range. Each generated GID was then used to select the associated Block Group polygon that was then buffered by a randomly generated value between a supplied minimum and maximum value (eg 1000m to 5000m). An example function header from PostgreSQL is as follows:

```
Create or Replace
Function BufferTests4326(p_loops      in integer default 100,
                         p_min_buffer in Numeric default 10,
                         p_max_buffer in Numeric default 5000)
Returns Boolean
…
```

The SQL used for the buffer comparison (for Google Mercator) are:

Oracle Spatial

```
select b.inCount,
       case when b.bGeom3785 is null
            then -1
            else sdo_util.getNumVertices(b.bGeom3785)
        end as outCount
  into v_inCount, v_outCount
  from (select sdo_util.getNumVertices(a.geom3785) as inCount,
               sdo_geom.sdo_buffer(a.GEOM3785,v_buffer_dist,0.05,'unit=M')
                 as bGEOM3785
          from PERF.M_BLOCKGROUP_2010 a
         where a.gid = v_gid
           and a.geom3785 is not null
        ) b;
```

PostGIS

```
select b.incount,
       case when b.bGEOM3785 is null
            then -1
            else ST_Npoints(b.bGEOM3785)
```

```
      end as result
  into v_inCount, v_outCount
  from (select ST_NPoints(a.GEOM3785) as inCount,
               ST_Buffer(a.GEOM3785,v_buffer_dist) as bGEOM3785
         from public.M_BLOCKGROUP_2010 a
        where a.gid = v_gid
          and a.GEOM3785 is not null
       ) b;
```

3rd party

```
select b.inCount,
       case when b.bGeom3857 is null
            then -1
            else SCHEMA.ST_NumPoints(b.bGeom3857)
        end as outCount
  into v_inCount, v_outCount
  from (select SCHEMA.ST_NumPoints(a.geom3857) as inCount,
               SCHEMA.ST_Buffer(a.GEOM3857,v_buffer_dist) as bGEOM3857
         from GIS.M_BLOCKGROUP_2010 a
        where a.gid = v_gid
          and a.geom3857 is not null
       ) b;
```

The results for the buffering of the Google Mercator data are:

Type	Average Vertices			Avg Time (sec)	Faster (x) than Slowest (1)
	In	Out	Diff		
3rd Party Type	308.76	294.62	-4.8%	0.0094	23.8
PostGIS	282.64	195.43	-44.6%	0.0371	6.0
Oracle Spatial	291.17	218.4	-33.3%	0.2235	1.0

Mercator Discussion

From the 100 tests, the average number of seconds to create each buffer is 23 times slower for Oracle, and 3 times for PostGIS that the fastest, the 3rd party. However, this figure does not reflect the utility of the operation. Is 22/100th of a second reasonable for an average buffering of a Cartesian object?
For the geodetic tests the following results were generated.

Type	Average Vertices			Avg Time (sec)	Faster (x) than Slowest (1)
	In	Out	Diff		
PostGIS	253.98	152.64	-66.4%	0.0433	5.1
Oracle Spatial	266.82	336.72	20.8%	0.2208	1

Geodetic Discussion

From the 100 tests, the average number of seconds to create each buffer is 5 times slower for Oracle than for PostGIS. However, this figure does not reflect the application of the operation. Is 22/100th of a second reasonable for the buffering of a geodetic object? When one considers such a metric does it make one less likely to use that function?

Appendix B: Loading 3rd Party Type Geometry via Parallel Chunking

The following script shows how the SDO_GEOMETRY data was copied across into the 3rd Party Type data whose constructor was exceedingly slow. The script shows how a procedure `g_building_insert` is created that accepts two input variables:

- p_start_id, and
- p_end_id.

These parameters will be supplied by the parallel chunking engine from the GID column of the PERF.G_BUILDING table. The procedure uses the range these values define to query for a **chunk** of data to convert. This chunk of data is converted from SDO_GEOMETRY to the 3rd Party Type's ST_GEOMETRY by the procedure (in parallel with others DBMS_PARALLEL_EXECUTE creates).

The last part of the script is an anonymous block that uses the DBMS_PARALLEL_EXECUTE API to:

- Create a named task ('g_building_task');
- Instructs DBMS_PARALLEL_EXECUTE to create a set of chunks (rows) of size 10,000 each from the PERF.G_BULDING table's GID column;
- Associates the procedure g_building_insert with the created chunks;
- Then runs the task with a potential parallel level (parallel procedures) of 16 (that is, if the table contains more than 16 * 10,000 rows – 160,000 – then a maximum of 16 parallel procedures could be executed at one);
- Finally, when the chunking finishes, the named task is deleted.

  The script, in total, is as follows.

```
DROP   TABLE GIS.G_BUILDING;
CREATE TABLE GIS.G_BUILDING (
  GID        INTEGER,
  CODE       NUMBER(4),
  FCLASS     VARCHAR2(40),
  NAME        VARCHAR2(100),
  MORTON_KEY INTEGER,
  GEOG4326   <SCHEMA>.ST_GEOMETRY
)
PCTFREE 0 PCTUSED 99 NOLOGGING;

CREATE OR REPLACE
PROCEDURE g_building_insert (p_start_id IN NUMBER, p_end_id IN NUMBER)
AS
  r_g_build c_g_build%ROWTYPE;
  CURSOR c_g_build IS
  SELECT GID,CODE,FCLASS,NAME,MORTON_KEY,GEOG4326
    FROM PERF.G_BUILDING c
   WHERE c.GEOG4326.get_gtype() = 3
     AND sdo_geom.validate_geometry(c.GEOG4326,0.005) = 'TRUE'
     AND c.GID BETWEEN p_start_id AND p_end_id;
BEGIN
  OPEN c_g_build;
  LOOP
    FETCH c_g_build INTO r_g_build;
    IF ( c_g_build%NOTFOUND ) THEN
      EXIT;
    END IF;
    BEGIN
      INSERT /*+APPEND */ INTO GIS.G_BUILDING (
                GID, CODE, FCLASS, NAME, MORTON_KEY, GEOG4326
      ) VALUES (
```

```
       r_g_build.GID, r_g_build.CODE, r_g_build.FCLASS, r_g_build.NAME,
r_g_build.MORTON_KEY,
       <SCHEMA>.ST_PolyFromWKB(r_g_build.GEOG4326.get_wkb(),4326)
      );
     COMMIT;
     EXCEPTION
       WHEN OTHERS THEN
         dbms_output.put_line('GID (' || r_g_build.GID || ') failed: ' || SQLERRM);
    END;
  END LOOP;
END g_building_insert;
/
SHOW ERRORS

set serveroutput on size unlimited
DECLARE
  l_task     VARCHAR2(30) := 'g_building_task';
  l_sql_stmt VARCHAR2(32767);
  l_try      NUMBER;
  l_status   NUMBER;
BEGIN
  BEGIN
    EXEC DBMS_PARALLEL_EXECUTE.drop_task(l_task);
    commit;
    EXCEPTION
      WHEN OTHERS THEN
        NULL; - doesn't exist
  END;
  DBMS_PARALLEL_EXECUTE.create_task (task_name => l_task);
  DBMS_PARALLEL_EXECUTE.create_chunks_by_number_col(task_name    => l_task,
                                                    table_owner  => 'PERF',
                                                    table_name   => 'G_BUILDING',
                                                    table_column => 'GID',
                                                    chunk_size   => 10000);
  l_sql_stmt := 'BEGIN g_building_insert(:start_id, :end_id); END;';
  DBMS_PARALLEL_EXECUTE.run_task(task_name      => l_task,
                                 sql_stmt       => l_sql_stmt,
                                 language_flag  => DBMS_SQL.NATIVE,
                                 parallel_level => 16);
  l_status := DBMS_PARALLEL_EXECUTE.task_status(l_task);
  dbms_output.put_line('Task_Status after result ='||l_status);
  DBMS_PARALLEL_EXECUTE.drop_task(l_task);
END;
/
SHOW ERRORS
```

Appendix C: Determining Suitable Method For Measuring Performance

Oracle has a number of methods for capturing the timing of the execution of a PL/SQL function. One is based on DBMS_UTILITY.GET_TIME while the other (based on timestamps) is as follows:

```
set serveroutput on size unlimited
Declare
   v_Start_Time TIMESTAMP(6);
   v_End_Time   TIMESTAMP(6);
   v_seconds    NUMBER;
   Function toSeconds(p_start in timestamp,
                      p_end   in timestamp)
   Return NUMBER
   As
   Begin
     Return extract(day    from (p_end-p_start))*24*60*60+
            extract(hour   from (p_end-p_start))*60*60+
            extract(minute from (p_end-p_start))*60+
            extract(second from (p_end-p_start));
   End toSeconds;
Begin
  for i in 1..10 loop
      v_Start_Time:= LOCALTIMESTAMP;
      dbms_lock.sleep(i);
      v_End_Time  := LOCALTIMESTAMP;
      v_seconds   := toSeconds(v_start_time,v_end_time);
      dbms_output.put_line(trim(TO_CHAR(v_seconds,'FM999990.9999990')));
  End Loop;
End;
/
show errors
```

While for PostgreSQL this method, also based on timestamps, was chosen:

```
Create or Replace Function check_timing()
Returns boolean
As
$BODY$
declare
   v_Start_Time Numeric;
   v_End_Time   Numeric;
   v_seconds    Numeric;
begin
  for i in 1..10 loop
    v_Start_Time:= extract(epoch from
                           cast(timeofDay() as timestamp))::Numeric;
    PERFORM pg_sleep(i);
    v_End_Time  := extract(epoch from
                           cast(timeofDay() as timestamp))::Numeric;
    v_seconds   := v_End_Time - v_Start_Time;
    RAISE NOTICE '%',trim(TO_CHAR(v_seconds,'FM999990.9999990'));
  End Loop;
  return true;
end;
$BODY$
  LANGUAGE plpgsql VOLATILE;

select check_timing();
```

These two methods were chosen as testing showed that they both returned equivalent results and so could be relied upon when timing tests.