Collaborative Project

GeoKnow - Making the Web an Exploratory Place for Geospatial Knowledge

Project Number: 318159 Start Date of Project: 01/12/2012 Duration: 36 months

Deliverable 2.4.1

Geospatial Clustering

Dissemination Level	Public
Due Date of Deliverable	Month 24, 30/11/2014
Actual Submission Date	Month 25, 06/01/2015
Work Package	WP2 – Semantics based Geospatial Information Management
Task	T2.4 – Geospatial Clustering
Type	Prototype
Approval Status	Final
Version	1.0
Number of Pages	28
Filename	D2.4.1_Geospatial_Clustering.pdf

Abstract: This document describes the implementation of geospatial clustering on RDF quads and characteristic sets.

Project funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.1	22/12/2014	Initial Draft	Hugh Williams
0.3	01/01/2015	Geospatial Clustering content	Orri Erling
0.4	02/01/2015	Reformatted Content	Hugh Williams
0.5	04/01/2015	Additional updates	Mirko Spasić
0.6	06/01/2015	Peer Review	Jens Lehmann
1.0	05/01/2015	Finalised Deliverable	Hugh Williams

Author List

Organisation	Name	Contact Information
OGL	Orri Erling	oerling@openlinksw.com
OGL	Hugh Williams	hwilliams@openlinksw.com
OGL	Mirko Spasić	mspasic@openlinksw.com
InfAI	Jens Lehmann	lehmann@informatik.uni-liepzig.de

Time Schedule before Delivery

Next Action	Deadline	Care of
First version	31/12/2014	Hugh Williams (OGL)
Second version	02/01/2015	Orri Erling (OGL)
Third version	04/01/2015	Mirko Spasić (OGL)
Final version	06/01/2015	Hugh Williams (OGL)

Executive Summary

This deliverable describes the prototype enhancements made to geospatial clustering support for rearranging physical storage according to geospatial criteria. It demonstrates benefits from reorganizing physical data placement according to geospatial properties. The use of Structure-Aware RDF Storage using characteristic set was also implemented and used in this deliverable. These techniques apply to both single servers and scale out elastic cluster Virtuoso configurations.

Abbreviations and Acronyms

Acronym	Explanation
BSP	Bulk Synchronous Processing
DFG	Distributed Join Fragment
DBMS	DataBase Management System
EWKT	Extended Well Known Text/Binary
ETL	Extract Transform Load
GeoSPARQL	standard for representation and querying of geospatially linked data for the Semantic Web from the Open Geospatial Consortium (OGC)
GPF	Graph Processing Framework
LGD	Linked Geo Data
LOD	Linked Open Data
MBR	Minimum Bounding Rectangle
NE	North-East
NW	North-West
OGC	Open Geospatial Consortium
OSM	OpenStreetMap (http://www.openstreetmap.org/)
OWL	Web Ontology Language
RDF	Resource Description Framework: the data model of the semantic web
RDFS	RDF Schema
SE	South-East
SPARQL	Simple Protocol and RDF Query Language: standard RDF query language
SQL	Structured Query Language: standard relational query language
SQL/MM	SQL Multimedia and Application Packages (as defined by ISO 13249-3)
SW	South-West
TPC-DS	Transaction Processing Council Decision Support benchmark
WGS84	World Geodetic System (EPSG:4326)
WKT	Well Known Text (as defined by ISO 19125)

Table of Contents

1. Introduction

This deliverable describes the prototype enhancements made to geospatial clustering support for rearranging physical storage according to geospatial criteria. It demonstrates benefits from reorganizing physical data placement according to geospatial properties. First the implementation of Structured-Aware RDF and Characteristic Set in Virtuoso will be described as this the Geospatial clustering enhancements and built on top of these features. These techniques apply to both single servers and scale out elastic cluster Virtuoso configurations.

The Virtuoso enhancements in this deliverable are available in the GIT v7fasttrack[1] *feature/emergent* branch.

2. Structure-Aware RDF Storage

The Virtuoso Structure-Aware RDF Store feature optimizes storage of RDF data in cases where the data exhibits regular, relational-like structure.

A Characteristic Set (CS) is a collection of subjects sharing several single-valued or nearly single-valued properties that occur together. For example, all triples extracted from a relational table naturally form a characteristic set.

2.1 Characteristic Set Schema

Each characteristic set is represented by a single table with a primary key of *S (Subject), G (Graph)*. All properties that are relatively dense and mostly single-valued are represented as dependent columns in the CS table. The high bits of the *S* identify the CS table. Missing property values are represented as NULLs and extra values are represented as entries in the *PSOG* index. The equivalent of a secondary index is an entry in the *POSG* index, which allows finding an *S* based on equality or range of the *O*. Not all predicates, nor all values in a predicate need have a *POSG* entry. There is for instance little point in a *POSG* entry for common values in a *P*. This often occurs with properties like rdfs:type or enumerations or measures like prices where the *O* is seldom a lookup key. An impression of having an index on everything is created at run time by actually having a *POSG* entry for selective values and by adaptively going to a scan for common values.

2.2 Characteristic Set Plan Generation

A query plan with characteristic sets starts with a SPARQL query expressed in terms of triple patterns. Triple patterns that have the same *S* and a *P* which is in at least one CS form abstract tables. These are treated like actual tables for purposes of query plan formulation. The abstract tables are often closely aligned with actual CS tables, making it possible to sample actual data for selectivity estimation. This is better than sampling at the triple pattern level since this can catch correlations between columns without actual joins. This also reduces the plan search space to the size of the relational equivalent, greatly accelerating compilation and allowing broader exploration of different plan shapes.

[1] https://github.com/v7fasttrack/virtuoso-opensource

Triple patterns that have an unspecified *P* or where the *P* is not in any CS are left as references to the RDF quad table. Plan generation involves index choice, i.e. whether to do a scan or a lookup on a possibly non-existent or partial *POSG* index for finding the *S*.

2.2.1 Run Time Adaptivity

The following cases of run-time adaptivity are supported:

- Abstract table scan: We start with a set of CS properties and filtering conditions on these. Some of the properties may be optional. The *S* may come from any CS in which all the non-optional properties occur. We note that occurrence here means either being a column in the CS or the existence of at least one *PSOG* entry with the *S* falling in the range of the CS. A special case is the *0* CS which does not have a table and corresponds to the default schema-less RDF index. For each such CS an executable sub-plan referring to the actual tables is constructed when first needed.

- The physical plan consists of the following scheme: Scan the CS table. For each row, look in RDF quads on PS for extra values. For each range of consecutive values found in the CS table, look at the first non-optional *P* in RDF quad so that the S was not in the CS table. This will capture matches that are not in the CS table, which may occur for example after deleting all but exception values.

When there is a non-satisfied condition on a column of the table, the *S* is not automatically disqualified since there could be a match in the exceptions. Too this effect, a bit vector is filled in. If the match depends on an exception, the corresponding RDF quad lookup has the meaning of inner join, else that of left outer join for the specific *S*.

- Lookup on *S* - The situation of matching a set of patterns where the S and optionally G are given is simpler. Each S uniquely identifies a CS. A match can however exist even if there is no row in the CS table, since all the properties may be found as exceptions.

A further complication occurs in both cases when there are many *G*'s for one *S*. A Cartesian product of all the *S/G/P* combinations must be generated to mimic the behavior of the RDF quad table. The need to do this is detected when processing the CS table: If there is a non-unique *S* in the result, the solutions where not each match has the same *G* (coming from the same row) are generated by another executable plan generated on demand. This is a join of single column lookups on the CS table, filtering out matches where all properties are from the same *G*. This is infrequent in practice, though.

When looking for *S* based on *O*, the plan usually involves a *POSG* lookup but in the case of known *P* and *O* values for which there is no *POSG*, this morphs into a no-op that passes a fleg to the next operation down the pipeline, which then morphs from a lookup by *S* into a scan of CS's looking for the *O*.

Finally, situations which are left as RDF quad references may still be matched from CS tables, depending on the run-time values of *S* and *P*. The RDF quad reference is therefore itself adaptive, cycling through a set of variations depending on the *S* and *P*.

The above range of adaptivity may appear expensive at first sight. In practice, exceptions are rare, and the more rare these are the less expensive they are to detect. For example, there is a Bloom filter for each *P* for the presence of an exception on a given *S*. If a table scan does has an unmatched condition

on a column and there is no match for the *S* in the Bloom filter of the *P* (column), then the *S* can be rejected there and then.

The CS logic is made even more efficient by vectored execution. All the exception checks for example take place on selection vectors, optimally exploiting pre-fetching and instruction level parallelism. The interpretation overhead of checking many different cases is de facto absorbed by doing this on vectors of tens of thousands of values. In principle heterogeneous (multi-cset) vectors may occur but with regular data this is uncommon and overheads are minimal.

The resulting executable plan is still expressed in terms of abstract tables. These are either table scans or lookups where the subject and possibly graph are known.

2.3 Data Clustering and Subject/Object Identifier Allocation

At data insertion time the RDF IRI's and literals concerned may or may not exist. If they exist, their identifiers are fixed and the CS membership of the subject IRI determines the CS, if any, that is affected. If the subject IRI does not have an identifier, one can be chosen from the available CS's depending on the IRI string, the properties and the property values present at the time.

The simplest case consists of having a sequence of ID's per CS. Additionally, property values may influence the choice of subject ID selection, so as to cluster subjects with similar property values together.

3. Geospatial Clustering

This section describes the implementation of geospatial clustering on RDF quads and characteristic sets (CS).

3.1 Selecting Clustering Grip Density

The identifiers of *O* and *S* for geometries and the subjects with geometry properties come from different ranges depending on the location of the geometry on the map.

In specific, there are 48 physical partitions spread across 4 server processes. 256 consecutive identifiers will be on one slice, the next 256 on the next slice and so forth.

The id range to use for geometries falling on a particular square is constructed from segments of 48 * 256 ids.

For each square, each process has an independent sliced sequence object for allocating id's for this square. There are in fact two sequences per square, one for the *O* id's for the geometry objects and one for the *S* id's of triples referencing the geometries.

We prefer to allocate the O and the S from the same slice. This means that these will, if newly allocated, be always co-located, saving on network communication. When a new geometry is seen, a hash is calculated for identifying the slice of its hash to id mapping (*RO_VAL* index of *RDF_OBJ*). The slice in question is consulted for checking if the identical geometry already exists. If this does not exist, an id has to be allocated. There is at this time a choice of per-square range and of slice. The square is

calculated at the start, together with the hash and is sent to the slice being checked. The slice of the *O* id is picked to be the same as that of the hash of the geometry. Now a square/slice specific id is obtained and associated to the hash.

At the same time, the *S* id of the triple might or might not be known. If the *S* IRI has an id, there is no further choice. If one must be allocated, it will be allocated from the same square and slice as the *O*. This is known independently of the *O* id, depending only on the geometry and the slice given by the geometry hash, so the eventual *S* allocation does not depend on results from other partitions and can proceed in parallel with the *O* allocation. The *S* must be checked for existence according in a slice given by a hash of the IRI string. This is nearly always different from the slice of the geometry hash. If an *S* id exists for the IRI of the *S*, there is no further choice. If this does not exist, then one can be selected from the slice of the geometry hash. To this effect, any process must be able to allocate id's in any slice. Therefore the id's do not come necessarily contiguous. This is not a problem however.

A square will due to the above have 256 (run of consecutive ids in per slice) * 48 (slices) * 4 (server processes) identifiers minimally associated with it. This is a minimum range of 49152 ids.

If this many geometries were evenly spread across all slices and all were accessed, this would come to 48 * 4 reads of 256 consecutive *O*'s and the same for the *S*'s. First the geometries get read, fetched from the R tree, leading to closely packed *O* values. Then the *O*'s get translated to *S*, then the *S* is used to retrieve property values associated to the subject with the geometry.

This would be realized if we split the world in squares of 48K geometries. This would make for the 2.6 billion geometries 53K squares. This is feasible but we will settle for a slightly coarser grain with 192K consecutive geometries per square. This comes to a read of 4 * 1024 consecutive places in each slice if the full square is accessed.

3.2 Geo Clustering With Characteristic Sets (CS)

Using Geo clustering with characteristic sets (CS) is similar to its use with triples. The characteristic sets situation uses an IRI pattern to bypass maintaining a name to id mapping table. However, since the ids directly derived from the string cannot be clustered, there must be a separate step for mapping the number derived from the string into a geo-clustered internal id. There is a table *RDF_N_IRI* for this mapping. This translates an external ID derived from the URI string into an internal one allocated from a per-square sequence and back.

We note that a number to number mapping is much more compact than a string to number mapping, specially with the user of column-wise compression.

There is a generic mechanism for declaring property-based clustering for characteristic sets. If clustering is to be used with a characteristic set, the subject identifier will be allocated from one of many "per-bucket" sequences, such that the sequence is selected based on the value of one or more clustering properties. A clustering property is declared in the declaration of the characteristic set itself, along with a clustering function and optional extra parameters. In these cases, there are multiple sub-sequences associated to the subject ID sequence of the characteristic set.

Id sequences thus form hierarchies, with a sequence having a densely numbered set of consecutive sub-sequences. These sub-sequences feed off the same sequence object as the super-sequence.

Such sub-sequences may exist under a property-clustered CS sequence but also under the sequence of literal ID's.

For example, in OSM there are two geo-clustered characteristic sets: *nodes_cset* and *ways_cset,* corresponding to the LGD/OSM nodes and ways. Since these are distinct characteristic sets these have disjoint ID ranges, differing the high bits of the *S* id.

Both are geo-clustered with the same grid of squares and hence both ID sequences have a sub-sequence for each Geo clustering grid square. The RDF literal id sequence itself has a top level geometry id sequence that feeds off the main sequence of object ids and in turn has a sub-sequence for each grid square.

The Geo clustering function, given the geometry itself, identifies the grid square. This gives the applicable sub-sequence under the applicable ID sequence. In the present case these are:

1. The per square literal object id's,
2. The nodes CS ids and
3. The ways CS id's.

The clustering function only indicates a bucket number, the position (which CS and whether this is a literal or IRI) determines the actual sequence to use.

As an example, the declaration of the nodes CS follows:

```
ir_init (name => 'geo_o_init', seq => 'RDF_RO_ID', txn_n_ways => 1, slice_bits => 8, n_slices => 48, chunk => 8);

ir_init_subs (super_name => 'geo_o_init', txn_n_ways => 1, n_subs => 5000, n_slices => 48, slice_bits => 8, chunk => 8);
```

These declare the per-square sequences for the geometry literal ids.

```
    rdf_cset ('nodes', properties => vector (
    'http://linkedgeodata.org/ontology/version',
    'http://purl.org/dc/terms/contributor',
    'http://purl.org/dc/terms/modified',
    'http://linkedgeodata.org/ontology/changeset',
    vector ('http://geovocab.org/geometry#geometry', 'index', 'cluster', 'cscl_geo', 'cluster_params', vector (1)),
    'http://www.w3.org/2003/01/geo/wgs84_pos#long',
    'http://www.w3.org/2003/01/geo/wgs84_pos#lat'),
    types => vector (), txn_n_ways => 1, n_slices => 48, slice_bits => 8, chunk => 8);
```

The above declares the CS, listing the properties that become columns. The geometry property is declared indexed and to be a clustering property for the CS. The index declaration means that for each value of this property in the CS there is a *POSG* and *OP* entry in *RDF_QUAD* for retrieving the *S* and *P* by the *O* or *PO*. If the value of the geometry property is given at load time and the *S* or the geometry *O* do not yet have a fixed id, then the clustering function can influence the allocation of the *S* and *O* ids. The clustering function suggests a preferred bucket/slice for the *O* and *S* and the ID allocation takes this into account if no ID exists yet.

The clustering functions are built in but the choice can be extended via a plug-in mechanism without recompiling the server executable.

```
cset_iri_pattern ('nodes', 'http://linkedgeodata.org/triplify/node%%', vector (vector (0, 10000000000)), int_range => ir_by_name ('DB.DBA.nodes_cset_rng'));
```

This declares the ID pattern for subject URI's for the CS.

```
ir_init_subs (super_name => 'DB.DBA.nodes_cset_rng', txn_n_ways => 1, n_subs => 5000,
n_slices => 48, slice_bits => 8, chunk => 8);
```

This declares a set of 5000 per-square sub-sequences for the main ID sequence created for the CS. The main sequence is implicitly created by the *rdf_cset* function.

All operations that create sequences either implicitly or explicitly take a set of common parameters for sizing the slices. These are

- **txn_n_ways** - Different transactions may feed off different sequences, so that concurrently allocated ids do not fall on the same page. This may improve insert performance by reducing contention on page latches. In the present case this is 1, so that all transactions get ids from the same sequence.

- **n_slices** - For an elsatic cluster, this is the number of distinct physical slices, in this case 12 slices in each of 4 server processes.

- **slice_bits** - This is the number of low bits that are ignored in slice calculation. 8 bits means that 256 consecutive ids fall on the same slice.

- **chunk** - When allocating a set of per-bucket (in this case per-sq square) ids, we must get at least n_slices << slice_bits worth of numbers, so that an id for any of the slices can be supplied from the set f numbers.

  For 448 slices * 256 consecutive ids per slices this would be 12288. In this case we allocate 8 times as many consecutive ids each time when getting a new set of ids for a square. When allocating a set of ids, the first id of the first slice always falls on slice 0, so that ranges are aligned on n_slices << slice_bits boundaries.

3.3 Data Load Behavior With Clustering

The dataset for the CS load was prepared by re-exporting the initially loaded OSM dataset so that triples of one subject were within predictable p[proximity of each other. The dataset produced by Sparqlify[2] is ordered by the property URI, which is ill suited for loading characteristic sets, especially with clustering. If only non-clustering properties are seen, the subjects must be assigned an ID for storage without the benefit of a clue for ID selection. Thus for these techniques to work the relevant properties of a subject must be seen together before there is a need to assign an ID to the subject for storage. Alternately, the data may be renamed in place, essentially reading one copy of the database while writing another.

The latter requires nearly double the space though and was not done here.

We note that there is no correlation between location geographical and OSM node or way identifier. Since all data dumps basically reflect this order, the clustering data load exhibits an insert behavior that is more scattered than that of a non-clustered load: Without clustering, ID's are assigned

[2] http://aksw.org/Projects/Sparqlify.html

consecutively, whereas with clustering these are assigned consecutively within any one of 4500 squares.

The clustering was selected so as to have squares of at most 2 million geometries. The division into squares was done on the basis of the non-clustered, triples dataset. The resulting number of squares was 4497.

These squares were first imported into the empty CS based database and were loaded in memory to serve as basis for the sequence selection of the *gscl* clustering function mentioned above in the declaration of the nodes *cset*.

The load was then started with 7 files * 4 processes worth of concurrent streams. The loading speed was initially 5.2M nodes per minute, falling to 2.2M nodes per minute after 1.2 billion nodes. Speeds of 4.8M nodes per minute were regularly observed after 1.5bn nodes. The variable locality of reference accounts for large variation in throughput. A node is between 10 and 15 triples.

The CPU profile for the load follows:

```
578696   12.3988  itc_geo_row
536841   11.5020  rd_box_union
121257    2.5980  dv_compare
98872     2.1184  cs_compress
97172     2.0819  gethash
80441     1.7235  ttlyylex
75365     1.6147  memzero
74166     1.5890  code_vec_run_1
66328     1.4211  itc_geo_check_link
62279     1.3344  id_hash_get
58996     1.2640  gen_qsort
53078     1.1372  dc_serialize_sliced
52858     1.1325  box_hash
52302     1.1206  pf_shift_compress
50302     1.0777  pg_row_check
```

The profile is dominated by maintaining the geo index, with the top 2 items making up 24% of the total. This is similar to previous results and the use of CS or clustering was not expected to change this. The 24% can be dropped to %around 5% by replacing external byte order floats with machine byte order ints for the R tree operations. This is a trivial change to be undertaken at such time when all opportunities with greater gains have been exhausted.

The loading working set appears as follows, at towards the end of the load, with 2.2bn geometries loaded:

```
KEY_TABLE                                                          INDEX_NAME
TOUCHES  READS    READ_PCT  N_DIRTY  N_BUFFERS

DB.DBA.RDF_GEO                                                       RDF_GEO
0        2017936  201793600  536335   1063323

DB.DBA.RDF_OBJ                                                       RO_VAL
252036859  1408331  0        1494235  2089006
```

```
DB.DBA.nodes_cset                                                    nodes_cset
16350745542  1214189  0        815126   2673934

DB.DBA.RDF_N_IRI                                                     RDF_N_IRI
290897863  674714   0        35300    89077

DB.DBA.RDF_QUAD                                                      RDF_QUAD_POGS
2377260949  529211   0        534009   787775

DB.DBA.RDF_OBJ                                                       RDF_OBJ
239912320  278590   0        99213    228350

DB.DBA.RDF_QUAD                                                      RDF_QUAD
878933247  207721   0        292833   403516

DB.DBA.RDF_N_IRI                                                     RI_N_REV
232954626  132818   0        188214   347628

DB.DBA.RDF_QUAD                                                      RDF_QUAD_SP
606396844  27923    0        125752   158338
```

The 4th column is the count of disk reads, the last is the count of buffers and the second last is the count of dirty buffers. The view concerns one of the 4 processes in the setup, the processes all have very similar state.

We note that the RDF Geo R tree counts for most of the reading and space utilization.

3.4 Space Utilization

Below is a summary of the space utilization in the Geo-clustered CS based database and the corresponding quads based one.

The first table is the row-wise structures in MB per each index. The second table gives the column-wise structures for each index, when present.

The tables are for the first server process of four, thus represent approximately a quarter of total space utilization. The relative sizes of the different structures are clearly visible. Some structures occur as both row and column, column-wise, as columnar structures have some row-wise superstructure also.

Row-wise:

```
DB.DBA.RDF_OBJ                31803
DB.DBA.RDF_GEO                22789
RO_VAL                        19536
DB.DBA.RDF_N_IRI              11588
DB.DBA.RDF_IRI                1708
DB_DBA_RDF_IRI_UNQC_RI_ID     994
DB.DBA.VTLOG_DB_DBA_RDF_OBJ   442
RDF_QUAD_POGS                 47
DB.DBA.nodes_cset             31
DB.DBA.RDF_QUAD               13
```

Column-wise:

DB.DBA.nodes_cset	26446
RDF_QUAD_POGS	7286
DB.DBA.RDF_QUAD	3990
RI_N_REV	2556
RDF_QUAD_SP	1397
DB.DBA.ways_cset	924
RDF_QUAD_OP	419
RDF_QUAD_GS	361

The numbers for quads are:

Row-wise:

DB.DBA.RDF_OBJ	32326
DB.DBA.RDF_IRI	28062
DB_DBA_RDF_IRI_UNQC_RI_ID	27324
DB.DBA.RDF_GEO	24518
RO_VAL	20995

Column-wise:

RDF_QUAD_POGS	35929
DB.DBA.RDF_QUAD	30953
RDF_QUAD_SP	7151
RDF_QUAD_OP	5633
RDF_QUAD_GS	313

In total space consumption the CS based variant is about 76% of the size of the quads based one.

We note a difference in space for IRI to string mapping. Since much of this is by number to number mapping in the node and way URI's, we see most of the load on RDF_IRI move to RDF_N_IRI. The row-wise index is not much better than the row-wise number to string one. The column-wise reverse index (from mapped to numeric id) is however much smaller, see RI_N_REV.

4. Query Execution Analysis

In this section we review performance of different query plans under different data layouts.

The base query is the following:

```
    sparql select ?f as ?facet ?latlon ?cnt
    where
    {
      {
        select ?f ?x ?y max(concat(xsd:string(?a)," ",xsd:string(?o))) as ?latlon
count(*) as ?cnt
        where
        {
          {
            select ?f ?a ?o xsd:integer(20*(?a - 45.1361)/4.5) as ?y xsd:integer(40*(?o -
4.0278)/9) as ?x
            where
            {
              {
                ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f . filter (?f =
<http://linkedgeodata.org/ontology/Village> || ?f =
<http://linkedgeodata.org/ontology/PlaceOfWorship> ||
<http://linkedgeodata.org/ontology/Restaurant> ||
<http://linkedgeodata.org/ontology/Sport>)
              }
              ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?a ;
                 <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o .
              ?s <http://geovocab.org/geometry#geometry> ?p .
              filter(bif:st_intersects(bif:st_geomfromtext("BOX(4.0278 45.1361, 13.0278
49.6361)"), ?p))
            }
          }
        }
        group by ?f ?x ?y
        order by ?f ?x ?y
      }
    }
    ;
```

This returns the types of selected entities in a wide geographical area. The area in question has 202M geometries, accounting for 7.7% of all geometries in the database.

We consider three alternative plans:

1. One builds a Hash table from the subjects with the geometries, then probes this with the subjects with selected types.

2. The second works purely by Index, with the search starting with the geo index, then proceeding to the types.

3. The third is for a Characteristic Set (CS) based plan for the same query

The Query Execution Plans are list in *Appendix 6.1* below. Here we summarize the most important comparison between these three query executions in the following table:

Query plan	Time	CPU	Rnd	Seq	Same seg	Same pg
Hash	589630	1363%	9.97E+008	8.86E+009	99.51%	0.43%
Index	344657	2477%	1.20E+009	4.38E+008	96.68%	3.15%
CS	115343	2971%	5.52E+008	2.38E+008	92.34%	7.31%

Table 1: Comparison of different query plans

4.1 Benefits of Characteristic Sets (CS)

We notice that the CS plan with clustering has a faster execution time and significantly reduced network traffic as opposed to the quads based queries with and without hash join.

We further see that 70% of time is spent in the entirely avoidable block with decimal arithmetic. Vectored floating point arithmetic will reduce this time to near zero.

The reduction in network traffic is clear since the data placement is such that the geometry object and the subject it qualifies are most often co-located.

We can see this in the low time utilization of the DFG stage 2 which is an exchange that sends tuples cross partition when going from the geometry to the subject. In practice, very little cross partition traffic takes place, hence the fast execution time.

We consider the CPU profile:

```
2422210    9.5795  num_divide
1387026    5.4855  dv_compare
1229055    4.8608  box_to_any_1
955277     3.7780  box_deserialize_string
899979     3.5593  _num_multiply_int
642112     2.5395  cs_decode
611079     2.4167  dc_any_cmp
```

```
570249    2.2553  sslr_qst_get
490142    1.9385  itc_param_cmp
471083    1.8631  dk_free_tree
```

We see that most of the time goes in arithmetic and that there is no point in doing decimal arithmetic in the application anyway. The database operations themselves are low in the profile.

5. Conclusions and Future Work

The CS results presented here were done by clustering the data at load time. A conversion from a loaded database of quads into CS's will be tried in the future. The implementation is straightforward, a simple copy plus delete from the quads to the CS tables, together with renaming subjects ids and geometry objects.

Performance of CS's against quads will be measured on a broader variety of queries. The benefits of clustering are the most obvious when query plans start with a geometry lookup: After all, such a lookup is expected to hit a number of tightly packed ID ranges. The same locality gains are expected to be seen also when approaching the data via other dimensions since nearby content will often end up being accessed.

The IN predicate used in the present experiments is not optimal. A better "invisible hash join" based IN predicate exists and will be used in future experiments.

We have implemented geospatial data clustering and characteristic sets based RDF storage. The CS based storage model is above and beyond the original scope of the GeoKnow project. The great benefit of this became apparent after the first GeoKnow review in comparing SQL and RDF quads based query performance. After the present initial demonstration of CS benefits we expect to attain near parity with SQL while running schema-less RDF behind the scenes enhanced with CS technology.

6. Appendix

6.1 Query Execution Plans

Hash plan:

```
    {
    time   5.5e-08% fanout         1 input         1 rows
    time      0.11% fanout         1 input         1 rows
    { hash filler
    wait time      0.19% of exec real time, fanout         0
    QF {
    time   3.4e-06% fanout         0 input         0 rows
    Stage 1
    time         2% fanout 4.31428e+06 input        48 rows
    geo  3  st_intersects  (CONTAINS  (<tag  238  c  BOX2D(4.027800  45.136100,13.027800
49.636100)>) node on DB.DBA.RDF_GEO      0.12 rows
    t12.O
    time       3.9% fanout  0.976704 input 2.07085e+08 rows
    RDF_QUAD_POGS   2.3e+08 rows(t12.S)
     P =  ##geometry  ,  O = cast
    time      0.28% fanout   0.99712 input 2.02261e+08 rows
    Stage 2
    time     0.093% fanout         0 input 2.02261e+08 rows
    Sort hf 50    2.3e+08 rows(q_t12.S) -> ()

    }
    }
    Subquery 56
    {
    time    0.0018% fanout         1 input         1 rows
    { fork
    time   4.7e-07% fanout         1 input         1 rows
    { fork
    wait time        39% of exec real time, fanout         0
    QF {
```

```
time   0.00022% fanout         0 input         0 rows
Stage 1
time      0.57% fanout 1.18286e+06 input        48 rows
RDF_QUAD_POGS     1e+08 rows(t9.O, t9.S)
 inlined  P =  ##type
hash partition+bloom by 54 ()
time     0.046% fanout         1 input 8.95077e+08 rows
END Node
After test:
      0: if ( 0  =  1 ) then 4 else 22 unkn 4
      4: if ( 0  =  1 ) then 8 else 22 unkn 8
      8: one_of_these := Call one_of_these (t9.O,  #/Village ,  #/PlaceOfWorship )
      13: one_of_these := Call one_of_these (t9.O,  #/Village ,  #/PlaceOfWorship )
      18: if ( 0  < one_of_these) then 22 else 23 unkn 23
      22: BReturn 1
      23: BReturn 0
time       1.4% fanout   0.92966 input 8.95077e+08 rows
Stage 2
time       2.2% fanout  0.477258 input 8.95077e+08 rows
Hash source 50        0.12 rows(q_t9.S) -> ()

After code:
      0: t12.S :=  := artm t9.S
      4: BReturn 0
time       1.5% fanout  0.848928 input 4.27183e+08 rows
RDF_QUAD         1 rows(t10.S, t10.O)
 inlined  P =  ##lat  ,  S = t12.S
time        29% fanout         1 input 3.62647e+08 rows

Precode:
      0: temp := artm t10.O -  45.1361
      4: temp := artm  20  * temp
      8: temp := artm temp /  4.5
      12: BReturn 0
RDF_QUAD         1 rows(t11.O)
 inlined  P =  ##long  ,  S = t10.S
time        58% fanout  0.999999 input 3.62647e+08 rows
```

```
    Precode:
          0: _cvt := Call _cvt (<constant>, temp)
          5: temp := artm t11.O -  4.0278
          9: temp := artm  40  * temp
          13: temp := artm temp /  9
          17: _cvt := Call _cvt (<constant>, temp)
          22: QNode {
    time         0% fanout         0 input         0 rows
    dpipe
    t11.O -> __RO2SQ -> __ro2sq
    t10.O -> __RO2SQ -> __ro2sq
    }

          24: _cvt := Call _cvt (<constant>, __ro2sq)
          29: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
          34: _cvt := Call _cvt (<constant>, __ro2sq)
          39: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
          44:  rdf_concat_impl  :=  Call  rdf_concat_impl  (__rdf_sqlval_of_obj,  <c    >,
__rdf_sqlval_of_obj)
          49: BReturn 0
    Stage 3
    time      0.83% fanout         0 input 3.62647e+08 rows
    Sort (set_no, q_q_t9.O, q__cvt, q__cvt) -> (rdf_concat_impl, inc)

    }
    }
    wait time         0% of exec real time, fanout         0
    QF {
    time   0.00038% fanout    2396.6 input        48 rows
    group by read node
    (gb_set_no, t9.O, _cvt, _cvt, aggregate, aggregate)
    time    0.0099% fanout         0 input    115037 rows

    Precode:
          0: QNode {
    time         0% fanout         0 input         0 rows
    dpipe
    t9.O -> __RO2SQ -> __ro2sq
    }
```

```
      2: BReturn 0
Sort (__ro2sq, _cvt, _cvt) -> (t9.O, aggregate, aggregate)

time   1.8e-08% fanout         0 input         0 rows
 ssa iterator
time     2e-05% fanout     10000 input         1 rows
Key from temp (__ro2sq, t9.O, _cvt, _cvt, aggregate, aggregate)

time         0% fanout         0 input     10000 rows
 qf select node output: (__ro2sq, aggregate, aggregate, _cvt, _cvt, t9.O, set_no)
}
}
time   0.00034% fanout     10000 input         1 rows
  cl fref read
 output: (__ro2sq, aggregate, aggregate, _cvt, _cvt, t9.O, set_no)
order:  6   0   3   4

After code:
      0: f :=  := artm t9.O
      4: x :=  := artm _cvt
      8: y :=  := artm _cvt
      12: latlon :=  := artm aggregate
      16: cnt :=  := artm aggregate
      20: BReturn 0
time   1.7e-08% fanout         0 input     10000 rows
Subquery Select(f, x, y, latlon, cnt)
}

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
f -> __RO2SQ -> facet
}

      2: BReturn 0
time     2e-08% fanout         0 input     10000 rows
```

```
Select (facet, latlon, cnt)
}


 589630 msec 1363% cpu, 9.96987e+08 rnd 8.85933e+09 seq 99.5124% same seg
0.428192% same pg
 1352689 messages 28391 bytes/m, 40% clw
Compilation: 13 msec 0 reads 0% read 0 messages 0% clw
```

Index based plan:

```
{
time 0.0021% fanout 1 input 1 rows
{ fork
time 2.8e-07% fanout 1 input 1 rows
{ fork
wait time 33% of exec real time, fanout 0
QF {
time 1e-05% fanout 0 input 0 rows
Stage 1
time 0.8% fanout 2.15309e+06 input 48 rows
geo 3 st_intersects (CONTAINS (<tag 238 c BOX2D(4.027800 45.136100,13.027800
49.636100)>) node on DB.DBA.RDF_GEO 0.12 rows
t12.O
time 3.2% fanout 0.976704 input 2.07085e+08 rows
RDF_QUAD_POGS 2.3e+08 rows(t12.S)
 P = ##geometry , O = cast
time 0.25% fanout 0.906418 input 2.02261e+08 rows
Stage 2
time 3.7% fanout 2.11204 input 2.02261e+08 rows
RDF_QUAD 2 rows(t9.O, t9.S)
 inlined P = ##type , S = q_t12.S
time 0.028% fanout 1 input 4.27183e+08 rows
END Node
After test:
      0: if ( 0 = 1 ) then 4 else 22 unkn 4
      4: if ( 0 = 1 ) then 8 else 22 unkn 8
      8: one_of_these := Call one_of_these (t9.O, #/Village , #/PlaceOfWorship )
      13: one_of_these := Call one_of_these (t9.O, #/Village , #/PlaceOfWorship )
```

```
      18: if ( 0  < one_of_these) then 22 else 23 unkn 23
      22: BReturn 1
      23: BReturn 0
time         3% fanout  0.848928 input 4.27183e+08 rows
RDF_QUAD         1 rows(t11.S, t11.O)
 inlined  P =  ##long  ,  S = k_t9.S
time        25% fanout         1 input 3.62647e+08 rows

Precode:
      0: temp := artm t11.O -  4.0278
      4: temp := artm  40  * temp
      8: temp := artm temp /  9
      12: BReturn 0
RDF_QUAD         1 rows(t10.O)
 inlined  P =  ##lat  ,  S = t11.S
time        63% fanout  0.999963 input 3.62647e+08 rows

Precode:
      0: _cvt := Call _cvt (<constant>, temp)
      5: temp := artm t10.O -  45.1361
      9: temp := artm  20  * temp
      13: temp := artm temp /  4.5
      17: _cvt := Call _cvt (<constant>, temp)
      22: QNode {
time         0% fanout         0 input         0 rows
dpipe
t10.O -> __RO2SQ -> __ro2sq
t11.O -> __RO2SQ -> __ro2sq
}

      24: _cvt := Call _cvt (<constant>, __ro2sq)
      29: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
      34: _cvt := Call _cvt (<constant>, __ro2sq)
      39: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
      44:  rdf_concat_impl  :=  Call  rdf_concat_impl  (__rdf_sqlval_of_obj,  <c    >,
__rdf_sqlval_of_obj)
      49: BReturn 0
Stage 3
time      0.85% fanout         0 input 3.62647e+08 rows
```

```
Sort (q_t9.O, q__cvt, q__cvt) -> (rdf_concat_impl, inc)

}
}
wait time         0% of exec real time, fanout         0
QF {
time   0.00035% fanout    2396.6 input        48 rows
group by read node
(t9.O, _cvt, _cvt, aggregate, aggregate)
time     0.013% fanout         0 input    115037 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
t9.O -> __RO2SQ -> __ro2sq
}

      2: BReturn 0
Sort (__ro2sq, _cvt, _cvt) -> (t9.O, aggregate, aggregate)

time   1.3e-08% fanout         0 input         0 rows
 ssa iterator
time   1.9e-05% fanout     10000 input         1 rows
Key from temp (__ro2sq, t9.O, _cvt, _cvt, aggregate, aggregate)

time         0% fanout         0 input     10000 rows
 qf select node output: (__ro2sq, aggregate, aggregate, _cvt, _cvt, t9.O)
}
}
time    0.0003% fanout     10000 input         1 rows
  cl fref read
 output: (__ro2sq, aggregate, aggregate, _cvt, _cvt, t9.O)
order:  0   4   3

After code:
      0: f :=  := artm t9.O
      4: x :=  := artm _cvt
```

```
        8: y :=  := artm _cvt
        12: latlon :=  := artm aggregate
        16: cnt :=  := artm aggregate
        20: BReturn 0
    time   1.7e-08% fanout         0 input     10000 rows
    Subquery Select(f, x, y, latlon, cnt)
    }

    After code:
        0: QNode {
    time         0% fanout         0 input         0 rows
    dpipe
    f -> __RO2SQ -> facet
    }

        2: BReturn 0
    time     2e-08% fanout         0 input     10000 rows
    Select (facet, latlon, cnt)
    }


     344657  msec  2477%  cpu,  1.19924e+09  rnd  4.37993e+08  seq      96.6808%  same  seg
3.15432% same pg
     180690 messages     91351 bytes/m,        33% clw
    Compilation: 10 msec 0 reads         0% read 0 messages         0% clw
```

The hash plan is more efficient but the index based one has better platform utilization, hence is faster. Next we consider the CS based plan for the same query.

```
    {
    time     6e-08% fanout         1 input         1 rows
    Subquery 28
    {
    time     1e-07% fanout         1 input         1 rows
    { fork
    time     0.044% fanout         1 input         1 rows
    { fork
    wait time         0% of exec real time, fanout         0
```

```
    QF {
    time     5e-05% fanout         0 input         0 rows
    Stage 1
    time       4.1% fanout 3.95957e+06 input        48 rows
    geo  3  st_intersects  (CONTAINS  (<tag  238  c  BOX2D(4.027800  45.136100,13.027800
49.636100)>) node on DB.DBA.RDF_GEO         0 rows
    d_id
    time        11% fanout  0.998566 input 1.90059e+08 rows
    RDF_QUAD_POGS         1 rows(txs_s.S)
     P =  ##geometry  ,  O = k_d_id
    time      0.33% fanout  0.951228 input 1.89787e+08 rows
    Stage 2
    time        10% fanout   0.90775 input 1.89787e+08 rows
    DB.DBA.nodes_cset_cv_18   2.2e+09 rows(c19.S, c19.long, c19.lat)
     S = q_txs_s.S
    time        74% fanout  0.489229 input 1.72279e+08 rows


    Precode:
          0: temp := artm c19.long -  4.0278
          4: temp := artm  40  * temp
          8: temp := artm temp /  9
          12: _cvt := Call _cvt (<constant>, temp)
          17: temp := artm c19.lat -  45.1361
          21: temp := artm  20  * temp
          25: temp := artm temp /  4.5
          29: _cvt := Call _cvt (<constant>, temp)
          34: QNode {
    time         0% fanout         0 input         0 rows
    dpipe
    c19.long -> __RO2SQ -> __ro2sq
    c19.lat -> __RO2SQ -> __ro2sq
    }


          36: _cvt := Call _cvt (<constant>, __ro2sq)
          41: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
          46: _cvt := Call _cvt (<constant>, __ro2sq)
          51: __rdf_sqlval_of_obj := Call __rdf_sqlval_of_obj (_cvt)
          56:  rdf_concat_impl  :=  Call  rdf_concat_impl  (__rdf_sqlval_of_obj,  <c    >,
__rdf_sqlval_of_obj)
```

```
      61: BReturn 0
RDF_QUAD       1.2 rows(t14.O)
 inlined  P =  ##type  ,  S = c19.S
time      0.02% fanout         1 input 8.42838e+07 rows
END Node
After test:
      0: if ( 0  =  1 ) then 4 else 22 unkn 4
      4: if ( 0  =  1 ) then 8 else 22 unkn 8
      8: one_of_these := Call one_of_these (t14.O,  #/Village ,  #/PlaceOfWorship )
      13: one_of_these := Call one_of_these (t14.O,  #/Village ,  #/PlaceOfWorship )
      18: if ( 0  < one_of_these) then 22 else 23 unkn 23
      22: BReturn 1
      23: BReturn 0
time      0.69% fanout         0 input 8.42838e+07 rows
Sort (t14.O, _cvt, _cvt) -> (rdf_concat_impl, inc)

}
}
time   0.00063% fanout     94531 input         1 rows
group by read node
(t14.O, _cvt, _cvt, aggregate, aggregate)
time     0.039% fanout         0 input     94531 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
t14.O -> __RO2SQ -> __ro2sq
}

      2: BReturn 0
Sort (__ro2sq, _cvt, _cvt) -> (t14.O, aggregate, aggregate)

}
time     4e-05% fanout     10000 input         1 rows
Key from temp (t14.O, _cvt, _cvt, aggregate, aggregate)
```

```
After code:
      0: f :=  := artm t14.O
      4: x :=  := artm _cvt
      8: y :=  := artm _cvt
      12: latlon :=  := artm aggregate
      16: cnt :=  := artm aggregate
      20: BReturn 0
time   3.2e-08% fanout         0 input     10000 rows
Subquery Select(f, x, y, latlon, cnt)
}

After code:
      0: QNode {
time          0% fanout         0 input          0 rows
dpipe
f -> __RO2SQ -> facet
}

      2: BReturn 0
time   4.8e-08% fanout         0 input     10000 rows
Select (facet, latlon, cnt)
}


 115343  msec  2971%  cpu,  5.52188e+08  rnd  2.38351e+08  seq       92.343%  same  seg
7.3114% same pg
100956 disk reads, 277001 read ahead,   41.1771% wait
 19962 messages     27915 bytes/m,      0.34% clw
Compilation: 8 msec 0 reads         0% read 0 messages         0% clw
```

7. References

[1] Chong, E., Das, S., Eadon, G., Srinivasan, J. : An Efficient SQL-based RDF Query Scheme, http://www.nesc.ac.uk/talks/683/oracle_rdf_query_vldb_2005.pdf, VLDB Conference (2005)

[2] Wilkinson, K. : Jena Property Table Implementation - Second International Workshop on Scalable Semantic Web Knowledge Base Systems, http://www.hpl.hp.com/techreports/2006/HPL-2006-140.pdf (2006)

[3] Levandoski, J., Mokbel, M. : RDF Data-Centric Storage - ICWS '09 Proceedings of the 2009 IEEE International Conference on Web Services , ISBN: 978-0-7695-3709-2 (2009)