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 2.6.1

## Prototype of built-in complex geo problem solving

| Dissemination Level | Public |
|---|---|
| Due Date of Deliverable | Month 20, 30/08/2014 |
| Actual Submission Date | Month 25, 05/01/2015 |
| Work Package | WP2 – Semantics based Geospatial Information Management |
| Task | T2.6 – Geospatial problem solving |
| Type | Prototype |
| Approval Status | Final |
| Version | 1.0 |
| Number of Pages | 16 |
| Filename | D2.6.1_Prototype_of_Built-in_Complex_Geo_Problem_Solving.pdf |

Abstract: This deliverable demonstrates database resident route planning for use in complex queries, using synthetically generated road network data and the graph analytical processing implemented in the Virtuoso engine.

The information in this document reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

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

## History

| Version | Date | Reason | Revised by |
|---|---|---|---|
| 0.1 | 17/12/2014 | Initial Draft | Hugh Williams |
| 0.2 | 29/12/2015 | Route planning data | René Pietzsch |
| 0.3 | 01/01/2015 | Graph Analytics | Orri Erling |
| 0.4 | 02/01/2015 | Reformatted Content | Hugh Williams |
| 1.0 | 05/01/205 | Finalised Deliverable | Hugh Williams |

## Author List


| Organisation | Name | Contact Information |
|---|---|---|
| OGL | Orri Erling | oerling@openlinksw.com |
| OGL | Hugh Williams | hwilliams@openlinksw.com |
| brox | René Pietzsch | rpietzsch@brox.de |


## Time Schedule before Delivery

| Next Action | Deadline | Care of |
|---|---|---|
| First version | 31/12/2014 | Hugh Williams (OGL) |
| Second version | 02/01/2015 | René Pietzsch |
| Third version | 04/01/2015 | Hugh Williams (OGL) |
| Final version | 05/01/2015 | Hugh Williams (OGL) |

# Executive Summary

This document describes a set of SQL extensions for graph analytics inside the Virtuoso database engine. Graph analytics typically consists of complex algorithms that make multiple passes over the dataset, generating large intermediate results. These are generally not expressible in a declarative query language. Procedural SQL is also not a poor fit for algorithms that are characterized by unpredictable random access patterns, besides procedural SQL tends to be interpreted, with the overheads this entails.

# 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 |
| 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 ....... 6
2. Complex Processing in Database Engine ....... 6
2.1 Single Source Shortest Path Example ....... 7
2.2 Temporary Table Options ....... 9
2.3 Group By INTO and Execute Clauses ....... 9
2.4 Transitive Block ....... 10
2.5 Aggregators ....... 10
2.6 Message Combination ....... 11
2.7 Parallelism and Synchronization ....... 11
3. Preparation of Route Planning Data ....... 12
4. Conclusions and Future Work ....... 14
5. References ....... 15

# 1. Introduction

This document describes a set of SQL extensions for graph analytics inside the Virtuoso database engine. Graph analytics typically consists of complex algorithms that make multiple passes over the dataset, generating large intermediate results. These are generally not expressible in a declarative query language. Procedural SQL is also not a poor fit for algorithms that are characterized by unpredictable random access patterns, besides procedural SQL tends to be interpreted, with the overheads this entails.

Due to this, graph analytics are typically done with specialized tools and frameworks, some of which also support scale-out, e.g. Giraph[1]. These are however disjoint from databases and require their own data staging and ETL and typically do not offer flexible ad hoc querying.

Therefore it is desirable to bring the graph analytics functionality within the DBMS, so as to have no special need for staging and ETL and so as to be able to build upon highly optimized distributed querying and message passing that is present in any advanced scale-out DBMS.

The Virtuoso enhancements in this deliverable are available in the GIT v7fasttrack[2] feature/emergent branch.

# 2. Complex Processing in Database Engine

We observe that the basic operations that make up a scale-out graph analytics framework are by and large already present in a scale-out DBMS. Adding graph analytics capabilities to a DBMS is therefore a matter of "unbundling" existing operators so that these can be used in novel combinations with better explicit control than that afforded by a declarative query language.

To this effect we introduce the following features:

- Named group by - Multiple statements may add to a single group by state and the aggregation can be user defined, consisting of arbitrary procedural logic. These group by's may be partitioned as any other table and therefore the partitioned group by control structure can be used as an exchange operator across threads/processes. These group by states can be scanned as tables or accessed by their grouping keys.

- BSP block - The basic idea of bulk synchronous processing or "GAS" (Gather, Apply, Scatter) is to run a piece of code on a set of items, such that each item generates an update that will be processed by the related items in the next step. In database terms, this is a table scan with filtering followed by a partitioned group by. The result of the group by is then the new state to be scanned by the next step. These steps may repeat until an arbitrary termination condition is reached.

- Table expression for effect - A BSP block can perform arbitrary queries, including queries that do

[1] http://dbpedia.org/resource/Apache_Giraph

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

cross partition data access. These queries produce a result that adds to the input of the next processing stage. Thus the queries do not per se make a result set but rather update the state of a named group by which will be scanned by a subsequent step. An Arbitrary number of named group by's and distinct processing stages can be combined to form a superstep. A superstep is the basic unit of iteration in BSP algorithms. All these data and control structures transparently scale from a single server multithreaded situation to a shared nothing cluster. All message passing and synchronization and flow control between threads is handled by the engine.

## 2.1 Single Source Shortest Path Example

The prototypical graph algorithm is the single source shortest path (SSSP) which takes a starting point in a graph and updates the state of all reachable vertices to be the shortest path distance form the starting vertex.

We consider a graph of weighted edges, for example a road network:

```
create table SP_VERTEX (sp_id bigint primary key, sp_score int);
alter index sp_vertex on sp_vertex partition (sp_id int (0hexffff00));

create  table  SP_EDGE  (spe_from  bigint,  spe_to  bigint,  spe_weight  int,  primary  key
(spe_from, spe_to) column);
alter index sp_edge on sp_edge partition (spe_from int (0hexffff00));

create table sssp_res (sr_id int primary key, sr_dist float, sr_head any_array);

create procedure sssp (in start int)
{
   declare sssp_state table (v_id int, primary key (v_id), v_dist float for transitive,
v_head  any  for  transitive,  border_FLAG  INT  FOR  TRANSITIVE,  t_border)  partition  (v_id
int);
  declare is_any any;
  is_any := aggregator ();
  group by  start into  sssp_state execute { v_dist := 0; is_border := 1; };

  transitive { agg_set (is_any, 0); }
  from sssp_state as start {
    if (border_flag)
    from sp_edge where spe_start = v_id group by  spe_to into sssp_state execute {
      if    (sssp_state.v_dist    is    null    or    edge.spe_weight    +    start.v_dist    <
sssp_state.v_dist) {
      sssp_stat.v_dist := start.v_dist + edge.spe_weight;
      sssp_state.border_flag := 1;
      sssp_state.v_head := start.v_id;
      agg_set (is_any, 1);
      }
    };
  }
  while (1 = agg_value (is_any));
  insert into sssp_res select v_id, v_dist, v_head from sssp_state;
}
;
```

The above procedure populates the *sssp_res* table with the set of vertices reachable from the start vertex, with the weight of the shortest path from start to each vertex. Each vertex additionally has the id(s) of the previous vertices on the shortest path(s). It is possible to have many paths of identical

length.

The procedure declares the *sssp_state* temporary table. This is initialized with a single row, this being the start vertex. An aggregator is declared to serve as a global flag for termination of the algorithm. An aggregator is an object which carries a state that is updateable from any partition or thread and that has a definite state at the end of a superstep. This state is typically some reduction of the updates received during the superstep. A minimum or boolean or (as in the above) are examples of reductions.

The temporary table declares all columns except the key to have two states (for transitive), the past state and the future state (to be the past state on the next superstep). Additionally the temporary table has the option *t_border* which adds a bit field to each row to indicate that rows with the bit set are the ones that will participate in the next superstep of the transitive block.

In the present example, the selection of active vertices is done with an explicit column of the temporary table but the *t_border* flag could have served also. The actual work is done by the FROM (top level table expression) in the transitive block. This takes the edge starting at the current row and gets the end point and cost. The group by then executes at the end point of the edge, with the future state of the edge's end vertex in scope. Since this is a group by, the vertex record in the *sssp_state* temp table is implicitly created when seeing a new grouping key.

We note that all execution is vectored but that the group by execute code needs to be run so that the vector of groups does not have duplicates. Updates of the same target vertex need to occur in some serializable order as the next one depends on the minimum cost set by the previous one.

The while below the from clause in the transitive block declares an end condition. If the aggregator was not set to 1, the state of the BSP evaluation did not change and there is no next state, hence the transitive block can terminate. An implicit termination is possible using the *t_border* flag: If no border exists, the evaluation is implicitly stopped.

When a transitive block stops, the last future state, for temp table rows that have such, becomes the past state and all other rows keep their previous state.

## 2.2 Temporary Table Options

A temporary table is declared with the declare statement in a procedural block. The temporary table can be updated with a named group by and can be referenced in a from clause of a table expression.

```
temp_table ::=
DECLARE  table_name  AS  (<base_table_element>[,  ...]  )  [PARTITION  [CLUSTER  cluster]
(<partition_spec>, ...)]

base_table_element ::=
column_name data_type {option} *
| PRIMARY KEY (<column> [, ...])
| t_border


option ::=
FOR TRANSITIVE
| NOT NULL
| DEFAULT <literal>
```

The table definition elements have their customary SQL meaning. The primary key for a temporary table is a hash table similar to that made by a group by. The primary key declaration is mandatory and defines the number and type of grouping keys which all must be supplied when inserting into the temporary table with group by named statements.

The *for transitive* option causes the column to exist in two copies so as to implement distinct past and future states of the column. If the column is referenced in a transitive block, the scan sees the past state and an eventual group by execute can read and write the future state that will serve as the past state in the next superstep.

The *t_border* option declares that the temporary table has a border flag built in. This is a bit field that can be set and controls whether a row participates in the next superstep.

When a temporary table occurs in a *FROM* clause, a scan of the temporary table is generated, unless there is a full match of all the grouping columns by equality, in which case the generated operator is a hash join probe. A reference to a temporary table in a group by execute statement accesses the table and executes a code block on a vector of rows of the temporary table.

## 2.3 Group By INTO and Execute Clauses

The group by clause in a query has a variant which aggregates into a named temporary table. The aggregation may be a set of customary aggregation functions of a procedural block. The procedural block has full rows of the temporary table in scope and can contain conditional assignments back into the temporary table.

```
GROUP  BY  <expression>  [,  ...]  INTO  temp_table  [AS  correlation_name]  EXECUTE  {
procedure_statement [; ...]}
```

## 2.4 Transitive Block

The transitive block is a parallel control structure that consists of scans of temporary tables where a vectored code block is run on consecutive rows of the each temporary table. It is assumed that the order of processing of the rows is not important and that the rows do not interact with each other, in other words is is assumed that vectored execution is possible.

```
TRANSITIVE init_code  trans_block [...]

init_code ::= compound_statement

trans_block ::=
FROM temp_table [AS correlation_name] { procedure_statement [...] }
  [WHILE search_condition]
```

The code block may contain top level table expressions which perform arbitrary queries and optionally end in a group by execute clause. The group by execute clause may be used for updating the state of a temporary table, including the state of the temporary table being scanned in the containing code block.

Columns of a temporary table which are declared for transitive will show their future value in the group by execute block and their past state in the transitive scan block. Correlation names may be used for distinguishing between references to new and old values. Both the *INTO <temp_table>* and the *FROM <temp_table>* in group by and transitive accept an optional correlation name.

An assignment to a column name of a temporary table inside the transitive code block or the group by execute code block will directly assign the referenced column in the temporary table.

A while clause can follow the from block. This is evaluated on a single state, which should be a summary (aggregation) of the complete parallel/distributed evaluation of the From clause above. This may for example read an aggregator or the superstep number and decide whether to stop. If the condition evaluates to false the whole transitive statement terminates.

The variable superstep is implicitly declared by the transitive block. It is 0 on the first invocation of each of the from blocks, 1 on the next and so forth. This may be used as a stopping condition in a while clause. If for example even and odd-numbered supersteps are meant to do a different thing, e.g. expand a perimeter left to right and right to left in a bidirectional search, then such a structure is best expressed as multiple from blocks and not as a condition on the superstep number.

If the *t_border* flag is present in the table, then an implicit termination takes place if no row has this flag set. In this way a while and aggregator is not always necessary for expressing an end condition.

## 2.5 Aggregators

An aggregator is an object that is in scope in all threads/processes participating to the evaluation of a transitive block. It may be used for keeping track of a global state of the computation. This is generally a reduction, e.g. minimum, e.g. the shortest path length found so far, or a flag or counter, e.g. how many solutions have been found.

An aggregator's state evolves during a superstep and can be consulted at any point inside a superstep.

For meaningful semantic the behavior of the aggregator should be monotonic. An aggregator is the only way in which vertices can communicate within a superstep. This communication is meaningful in optimization tasks, e.g. shortest paths, since trying a path that is longer than the shortest so far makes no sense. The situation may evolve and the evolution may be taken into account within the superstep itself.

The general form of the aggregator is a *top k order by*. This keeps a set number of values of a measure and allows adding new ones and reading any of the values. The states are shared between threads and processes in an asynchronous but low-latency manner, so that timely values are always available.

The same mechanism is used in query evaluation for optimizing a *top korder by*, e.g. if the top 10 largest orders are to be found and there are already 10 orders found, then orders smaller than the 10th need not be considered since these will not make the result. This provides a cutoff that can be pushed down in query evaluation. The same mechanism serves for implementing aggregators.

## 2.6 Message Combination

A single end vertex may be reached a large number of times during a superstep. Some of the edges going to the same destination may originate in the same vector worth of source vertices. This offers the possibility of pre-combining these messages, so that only the resultant of the two is actually sent to the target partition.

This feature is sometimes found in BSP frameworks. The applicability of message combining greatly depends on the algorithm being implemented.

In Virtuoso, message combining is not presently supported but can in the future be easily added as an option to the group by execute clause. In this case, before taking the cross partition step, a reduction can be applied to the vectors going to the partition exchange: If two rows in the vector go to the same place, these may be collapsed into one by an application dependent function.

## 2.7 Parallelism and Synchronization

The basic building block of distributed execution in Virtuoso is the DFG (Distributed Fragment) described in LOD2[3] project deliverable D2.6[4] This is a pipeline of multiple stages, where each stage involves *n* producers and *m* consumers where each producer may send input to any consumer according to a partitioning key. If there is no partitioning key, the output of all producers may also be processed by all consumers. The producer is typically a table scan or a join and the consumer is a join against a partitioned table where the join keys depend on a previous stage of the DFG.

The scan at the start of a transitive block is the first stage of a DFG. A piece of procedural code is executed on a vector of rows produced by the scan. These are all independent, hence vectored execution is appropriate, each row of the scan belongs to a different grouping key (vertex). The blocks may contain a query for effect with a partitioned group by execute. The query for effect is run on a

---

[3] http://lod2.eu

[4] http://static.lod2.eu/Deliverables/D2.6_Knowledge_Store_Release_With_Integrated_Bulk_Processing_Features.pdf

whole vector of vertices, typically for the purpose of retrieving connected vertices. The group by execute block is then executed for the future state of the connected vertex. This is done serially within each partition, since multiple edges may terminate in the same place and the resulting updates to the state of the end vertex need to happen in some serial order.

For parallelism, there is one thread per partition at all times. Thus the initial scan is done at full platform and the partitioning crossing group by's are handled by the same thread as handles the scan of the partition in question. There is no lock or mutex controlling access to any part of the group by state. Each partition of the group by may also grow (rehash) independently. The only serialization occurs in the DFG operator which handles message passing between threads and this is highly optimized.

By having more partitions than hardware threads one may build some tolerance against partition skew which is inevitable in graph applications.

The single server and scale-out variants of the operation work in a similar way, except that the cluster variant additionally deals with inter-process message passing.

For a superstep of a transitive block, the temporary table being read is expected to be partitioned, so that there is a parallel thread per partition. A table expression inside the parallel block will execute on a vector of rows. The table expressions are directly nested inside the DFG whose first stage is the scan of the transitive temp table. In this way, each per-partition thread is both a producer and consumer of group by named states.

# 3. Preparation of Route Planning Data

Road network data need to be prepared for demonstrations and scenarios on built-in complex geo problem solving. Therefore a simple vertices / edge based road data model had to be derived from the OpenStreetMap (OSM) raw data.

The OSM data model only has three primitive *Elements*[5]:

1. Node,
2. Way and
3. Relation.

Each of them can have *Tags* that describe the context in which they are used and how their values have to be interpreted and rendered on a map.

Nodes describe a specific point on the surface of the earth by its longitude and latitude values.

Ways are typically used to describe linear features such a rivers, borders / boundaries or roads. Ways carry an ordered list of nodes that are referenced. If two (or more) ways (highways) share the same node it means that there exists an intersection / crossing, which mostly means that you can turn from one road into another (neglecting turn restrictions, one way streets, etc.).

As one specific OSM way may comprise multiple nodes and as such can act in crossings, ramps, merges, etc. route planning and path calculation becomes complex and expensive based on this data

[5] http://wiki.openstreetmap.org/wiki/Elements

representation. Thus we transformed the raw data to a much simpler model. Therefor we first shrink down the amount of data to the relevant, selecting the OSM raw data of Europe as starting point for the data transformation. Second we filtered out the road network information from the generic node and way data. OSM classifies different highway level[6] and we selected motorway, trunk and primary level (including respective link roads) to be included into our road network data.

The prepared dataset is extremely simple: containing vertices (points table) and edges (edge table) only. From the available OSM metadata we selected the following subset, relevant of capacity calculations:

- highway – the type of road according to the OSM highway level classification
- lanes – the number of road lanes
- maxspeed – the effective speed limit
- oneway – indicating the current segment to be a one-way street (direction indicated from start_point to end_point)

The following figure shows the resulting ERM model:

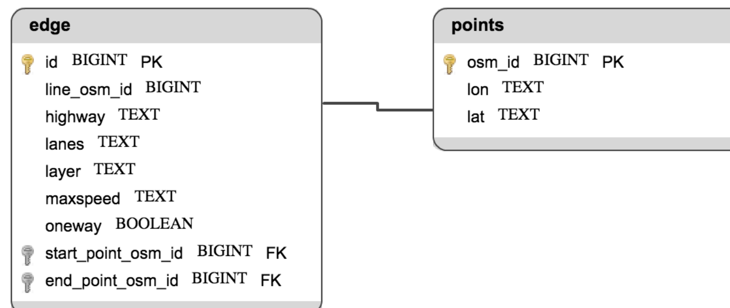


**Figure 1: ERM model of the road network data.**

[6] http://wiki.openstreetmap.org/wiki/Map_Features#Highway

# 4. Conclusions and Future Work

The Virtuoso BSP extension will be used for implementing a range of graph algorithms as part of specifying and implementing the LDBC graph analytics workload. This will provide extensive performance metrics for this capability above and beyond geospatial applications.

The present work for the first time integrates a graph processing framework into a general purpose scale-out DBMS. We see that there is no great impedance mismatch between procedural SQL and a bulk synchronous processing framework. We see that all the core components of a BSP framework are in fact found in a full feature parallel DBMS like Virtuoso.

The advantages of bringing processing close to the data are obvious:

- There is no distinct ETL step and updates are handled buy the DBMS as a matter of course. The data representation is already highly optimized and scale-out capable, benefiting from the whole range of column store and vectored execution techniques.

- The novel opening explored here may play a role in shaping the GDB and GPF (graph processing framework) scenes, specially via the exposure this gets in GeoKnow's sister project LDBC[7].

[7] http://ldbcouncil.org

# 5. References

[1] Chafi, H., Welc, A., Raman, R., Wu Z., Hong, S., Banerjee, J.: Graph Analysis – Do We Have to Reinvent the Wheel? GRADES'13: First International Workshop on Graph Data Management Experiences and Systems, ISBN: 978-1-4503-2188-4 (2013)

[2] Chafi, H., Hong, S., Sedlar, E., Olukotun, K.: Green-Marl: A DSL for Easy and Efficient Graph Analysis, http://www.cl.cam.ac.uk/~ey204/teaching/ACS/R202_2012_2013/papers/S7_Network_Structure/papers/hong_asplos_2012.pdf, ACM978-1-4503-0759-8/12/03 (2012)

[3] Thompson, B., Personick, M., Fu, Z.: MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs, GRADES'14, http://mapgraph.io/papers/MapGraph-SIGMOD-2014.pdf, ACM 978-1-4503-2982- 8/14/06 (2014)

[4] Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. The Semantic Web – ISWC 2010 , http://link.springer.com/chapter/10.1007%2F978-3-642-17746-0_48, ISBN 978-3-642-17746-0 (2010)

[5] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski: Pregel: A System for Large-Scale Graph Processing. http://kowshik.github.io/JPregel/pregel_paper.pdf , SIGMOD'10, ACM 978-1-4503-0032-2/10/06 (2010)