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 2.3.1

Prototype of built-in Geospatial Capabilities

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Confidential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 12, 30/11/2013</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>30/11/2013</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP2 – Semantics based Geospatial Information Management</td>
</tr>
<tr>
<td>Task</td>
<td>T2.3 – Geospatial Query Optimisations</td>
</tr>
<tr>
<td>Type</td>
<td>Prototype</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Final</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>50</td>
</tr>
<tr>
<td>Filename</td>
<td>D2.3.1_Prototype_of_Built-in_Geospatial_Capabilities.pdf</td>
</tr>
</tbody>
</table>

Abstract: The purpose of this deliverable is to present a first prototype of the geospatial enhancements possible and those that have been implemented in Virtuoso, the RDF store for GeoKnow generator.

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.
## History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>23/11/2013</td>
<td>Initial Draft</td>
<td>Hugh Williams</td>
</tr>
<tr>
<td>0.1</td>
<td>24/11/2013</td>
<td>Virtuoso Query Optimisations</td>
<td>Orri Erling</td>
</tr>
<tr>
<td>0.2</td>
<td>27/11/2013</td>
<td>Estimating Selectivity of Spatial Predicates</td>
<td>Kostas Patroumpas</td>
</tr>
<tr>
<td>0.3</td>
<td>28/11/2013</td>
<td>Reformatting of content</td>
<td>Kostas Patroumpas</td>
</tr>
<tr>
<td>0.4</td>
<td>28/11/2013</td>
<td>Updated content</td>
<td>Hugh Williams</td>
</tr>
<tr>
<td>0.5</td>
<td>29/11/2013</td>
<td>Updated content</td>
<td>Hugh Williams</td>
</tr>
<tr>
<td>1.0</td>
<td>30/11/2013</td>
<td>Final Cleanups</td>
<td>Hugh Williams</td>
</tr>
<tr>
<td>1.0</td>
<td>04/12/2013</td>
<td>Peer Review</td>
<td>Saleem Muhammad</td>
</tr>
</tbody>
</table>

## Author List

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>OGL</td>
<td>Hugh Williams</td>
<td><a href="mailto:hwilliams@openlinksw.com">hwilliams@openlinksw.com</a></td>
</tr>
<tr>
<td>OGL</td>
<td>Orri Erling</td>
<td><a href="mailto:oerling@openlinksw.com">oerling@openlinksw.com</a></td>
</tr>
<tr>
<td>ATHENA</td>
<td>Kostas Patroumpas</td>
<td><a href="mailto:kpatro@dblab.ece.ntua.gr">kpatro@dblab.ece.ntua.gr</a></td>
</tr>
<tr>
<td>ATHENA</td>
<td>Spiros Athanasiou</td>
<td><a href="mailto:spathan@imis.athena-innovation.gr">spathan@imis.athena-innovation.gr</a></td>
</tr>
<tr>
<td>ATHENA</td>
<td>Giorgos Giannopoulos</td>
<td><a href="mailto:giann@imis.athena-innovation.gr">giann@imis.athena-innovation.gr</a></td>
</tr>
</tbody>
</table>

## Time Schedule before Delivery

<table>
<thead>
<tr>
<th>Next Action</th>
<th>Deadline</th>
<th>Care of</th>
</tr>
</thead>
<tbody>
<tr>
<td>First version</td>
<td>23/11/2013</td>
<td>Hugh Williams (OGL)</td>
</tr>
<tr>
<td>Second version</td>
<td>27/11/2013</td>
<td>Kostas Patroumpas (ATHENA)</td>
</tr>
<tr>
<td>Third version</td>
<td>28/11/2013</td>
<td>Hugh Williams (OGL)</td>
</tr>
<tr>
<td>Final version</td>
<td>30/11/2013</td>
<td>Hugh Williams (OGL)</td>
</tr>
</tbody>
</table>
Executive Summary

An analysis of possible Geospatial query optimisation that could be implemented in RDF query optimizers to choose the best execution plans for queries is presented. This focuses on estimating the selectivity of spatial predicates which is an important and well-studied problem in relational DBMSs.

A detailed description of the Geospatial enhancements in supported datatype and query optimisations made in Virtuoso is presented. Query plans for scale-out execution of Geobench queries are analyzed and the time consumed in each operator is discussed. A separate section discusses impact of optimized data placement on query evaluation performance. The source code and prototype server for demonstrating the enhancements made is also made available.
### Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFG</td>
<td>Distributed Join Fragment</td>
</tr>
<tr>
<td>DBMS</td>
<td>DataBase Management System</td>
</tr>
<tr>
<td>EWKT</td>
<td>Extended Well Known Text/Binary</td>
</tr>
<tr>
<td>GeoSPARQL</td>
<td>standard for representation and querying of geospatially linked data for the Semantic Web from the Open Geospatial Consortium (OGC)</td>
</tr>
<tr>
<td>LOD</td>
<td>Linked Open Data</td>
</tr>
<tr>
<td>MBR</td>
<td>Minimum Bounding Rectangle</td>
</tr>
<tr>
<td>NE</td>
<td>North-East</td>
</tr>
<tr>
<td>NW</td>
<td>North-West</td>
</tr>
<tr>
<td>OGC</td>
<td>Open Geospatial Consortium</td>
</tr>
<tr>
<td>OSM</td>
<td>OpenStreetMap (<a href="http://www.openstreetmap.org/">http://www.openstreetmap.org/</a>)</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework: the data model of the semantic web</td>
</tr>
<tr>
<td>RDFS</td>
<td>RDF Schema</td>
</tr>
<tr>
<td>SE</td>
<td>South-East</td>
</tr>
<tr>
<td>SPARQL</td>
<td>Simple Protocol and RDF Query Language: standard RDF query language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language: standard relational query language</td>
</tr>
<tr>
<td>SQL/MM</td>
<td>SQL Multimedia and Application Packages (as defined by ISO 13249-3)</td>
</tr>
<tr>
<td>SW</td>
<td>South-West</td>
</tr>
<tr>
<td>WGS84</td>
<td>World Geodetic System (EPSG:4326)</td>
</tr>
<tr>
<td>WKT</td>
<td>Well Known Text (as defined by ISO 19125)</td>
</tr>
</tbody>
</table>
# Table of Contents

1. **Introduction** ......................................................................................................................... 8  
   1.1 Towards Spatial Histograms for RDF Query Optimisation .................................................. 8  
   1.2 Virtuoso Geospatial Enhancements ....................................................................................... 8  

2. **Estimating Selectivity of Spatial Predicates** ....................................................................... 10  
   2.1 Towards Spatial Histograms for RDF Query Optimisation ................................................... 10  
      2.1.1 Background .................................................................................................................. 10  
      2.1.2 Handling Spatial Predicates in Query Optimisation ...................................................... 11  
      2.1.3 Our Approach to Spatial Selectivity Estimates from RDF Datasets .............................. 12  
   2.2 **Grid-based Histograms** .................................................................................................... 13  
      2.2.1 Overview of Uniform Grid Partitioning ........................................................................ 13  
      2.2.2 Employing Grid-based Histograms for Spatial Selectivity Estimation ........................ 14  
   2.3 **Quadtree-based Histograms** ........................................................................................... 17  
      2.3.1 Overview of Standard Quadtrees .................................................................................. 17  
      2.3.2 Using MBR-Quadtrees as Spatial Selectivity Estimators ............................................. 18  
   2.4 **Data-driven Spatial Histograms** ....................................................................................... 20  
      2.4.1 Overview of the R-Tree Family .................................................................................... 20  
      2.4.2 Using Aggregate R-trees as Spatial Selectivity Estimators ........................................ 21  
   2.5 **Experimental Evaluation** .............................................................................................. 23  
      2.5.1 Use Case ...................................................................................................................... 23  
      2.5.2 Experimental Setup ..................................................................................................... 24  
      2.5.3 Experimental Results .................................................................................................. 25  
   2.6 **Outlook** ............................................................................................................................ 31  

3. **Virtuoso GeoSpatial Query Optimisation Enhancements** .................................................. 32  
   3.1 Data Types and Structures ................................................................................................. 32  
   3.2 Geometry Index ................................................................................................................... 33  
   3.3 SQL and RDF use of Geometries ....................................................................................... 33  
   3.4 Reading a Virtuoso Query profile plan .............................................................................. 34  
   3.5 Query Optimization ............................................................................................................ 37  
   3.6 Examples of Geospatial Query Plans ............................................................................... 37  
      3.6.1 Execution Plan Profiles ............................................................................................... 39  
   3.7 **Scale Out and Data Location** ........................................................................................ 45  
   3.8 Execution Parallelism ........................................................................................................ 45  
   3.9 System Prototype .............................................................................................................. 45  
   3.10 Conclusions and Future Work ......................................................................................... 47  

4. **References** .......................................................................................................................... 48
List of Figures

Figure 1: Example of a Cell Histogram based on 5x5 grid decomposition ........................................... 15
Figure 2: Example of a Centroid Histogram based on 5x5 grid decomposition ....................................... 16
Figure 3: Example of an Euler Histogram based on 5x5 grid decomposition .......................................... 17
Figure 4: Example of a MBR-Quadtree and its associated histogram for quadrant capacity c=4 .... 19
Figure 5: Example of an R-tree with at least m=2 and at most M=5 per node ........................................... 22
Figure 6: Illustration of the corresponding ar-tree histogram ...................................................................... 22
Figure 7: Data Universe and indicative query workload used in the evaluation study .............................. 24
Figure 8: Preprocessing cost of spatial histograms against the entire dataset ............................................ 26
Figure 9: Performance and accuracy of grid-based histograms ................................................................ 27
Figure 10: Performance and accuracy of quadtree-based histograms ...................................................... 27
Figure 11: Performance and accuracy of data-driven spatial histograms ................................................ 28
Figure 12: Comparison of spatial histograms for varying range sizes ...................................................... 29
Figure 13: Spatial indexing schemes with default settings over the OSM datasets of Great Britain 31

List of Tables

Table 1: Data contents of OSM layers (ESRI shapefiles) used in the evaluation ........................................ 23
Table 2: Experiment parameters ............................................................................................................... 25
Table 3: Geobench samples ...................................................................................................................... 38
1. Introduction

The deliverables is divided into two distinct sections:

- An estimation of selectivity of spatial predicates for RDF query optimisations, performed by Athena.
- A description of the Geospatial enhancement in query optimisations and new data types implement in Virtuoso, which is the RDF data store used in the GeoKnow project, performed by OpenLink for this prototype deliverable.

1.1 Towards Spatial Histograms for RDF Query Optimisation

We also examine several alternative techniques for spatial selectivity estimation in queries over geospatial RDF data. Compared to relational DBMSs, choosing a cost-effective execution plan is far more complex for queries with spatial predicates against large geographic datasets. Not only the geometric complexity of features (points, polylines, polygons, etc.), but also skewness and non-uniform density that are inherent in most geographical datasets, make this task very challenging. As the RDF data may be hosted in remote sites (e.g., exposed through different SPARQL endpoints), a query has to examine each one of them, collect intermediate results, and finally match them to produce the qualifying triples. So, having even a rough estimate about the amount of geometries that may qualify for topological predicates (e.g., contained or intersecting the area of interest) can offer significant savings in processing cost.

The core idea of our approach is to enhance well-known spatial indexing methods in order to quickly yield approximate, yet quite reliable, spatial selectivity estimates. We adjust variants of grid partitioning schemes, quadtrees, and R-trees towards spatial analogues of histograms. Essentially, rectangular regions of space are abstracted as "buckets" can report the cardinality of geometries therein. We stress that our focus is strictly on the spatial part of queries, as this poses the greater difficulty to assess. Our analysis and extensive empirical study in terms of performance and accuracy against typical query workloads provides strong evidence that such histograms could greatly assist in more advanced RDF optimization techniques.

1.2 Virtuoso Geospatial Enhancements

The Virtuoso commercial release has long included support for a geometric data type and corresponding R-tree index, based on the SQL/MM scheme as implemented by PostGIS and many other database. This includes optimiser support for geometry cardinality sampling and good execution plans for combinations of spatial and other joins, but was limited point data types. It has been decided to make this support available in the Virtuoso open source release and add support for the additional geometry data type and improve geospatial query optimisation as part of the GeoKnow project in WP2.

In this deliverable we present the first prototype of the enhanced geospatial support that has been added to Virtuoso, which includes support for additional data types and cost model optimiser
enhancements to provide reliable knowledge of data cardinalities and predicate selectivity and guarantee always getting a workable query plan also in the presence of unexpected data skew and correlations. This is achieved by the optimizer randomly evaluating sampled subsets of the data, thus detecting selectivity of joins with real data instead of pre-computed statistics, which can be used to devise an optimal query plan for execution.
2. Estimating Selectivity of Spatial Predicates

Selectivity estimation of queries is an important and well-studied problem in relational DBMSs. In this Section, we examine selectivity estimation in the context of geospatial RDF datasets, which may include not only points, but also more complex geometries such as lines, polylines and polygons. The core idea of our approach is to enhance well-known spatial indexing methods in order to yield a rough, but quick estimation of the amount of geometries that potentially qualify for typical topological predicates (e.g., contains, within, intersects, etc.).

In a nutshell, we believe that having a spatial analogue of histograms can be valuable when RDF query optimizers need to choose the best execution plans for queries involving both spatial and non-spatial predicates. We have extensively tested each spatial histogram against a workload of range queries of varying area sizes and we compared the fluctuations in their performance and accuracy. Our focus is strictly on the spatial part of the queries, as this poses the greater difficulty to analyse. Any method should be able to cope with the intricate properties inherent in most multidimensional datasets, predominantly skewness and non-uniform density. We deem that our analysis could greatly assist in more advanced RDF optimization techniques that may take into account spatial selectivities for reordering joins and choosing suitable query plans that also involve non-spatial criteria.

In the sequel, we first provide an insight about the benefits of fast and accurate selectivity estimation over spatial datasets, especially in the context of the RDF data management. Then, we analyse how it is possible to extend a few powerful state-of-the-art spatial access methods in order to get spatial selectivity estimates during query optimisation. We discuss several alternative data structures, in order to study their behaviour in terms of computational performance and estimation accuracy. Our detailed experimental study on real datasets provides insightful conclusions about the strengths and weaknesses of each method and their potential use by RDF query optimizers.

2.1 Towards Spatial Histograms for RDF Query Optimisation

2.1.1 Background

As in relational DBMSs, accurate estimates of query result sizes are very important in spatial data evaluation. In particular, query optimizers can use such estimates in order to determine the most efficient query plans. These estimates can be also helpful for accumulating statistics to approximate the underlying data distribution and assess result sizes based on these statistics.

Histograms, sampling, wavelets, sketches, and many other techniques [PIHS96, HNS94, MVW98, DGGR09] have been used in relational databases for selectivity estimation. Of the various techniques, histograms are the most popular [loan03], thanks to their computational efficiency and little space cost, and also because they do not require a priori knowledge of the input distribution. This scheme partitions the input into a small number of "buckets", and uses approximations to model the distribution of the respective subset of data records in each bucket.

As argued in [APR99], spatial selectivity estimation differs in two important aspects from traditional selectivity estimation:

a) The individual spatial entities in the dataset may differ in shape and size. For example, a dataset representing the islands of Greece will include some large polygons (e.g., Crete), as well as many smaller ones (like those in Cyclades), each with a very different coastline. Spatial objects often need very large geometric representations (hundreds or thousands of
coordinates), so it is quite inefficient to manipulate them directly for selectivity estimation.

b) The distribution of frequencies over the input domain does not vary significantly in spatial data, but the values are spread non-uniformly in space. For instance, there may not be many locations in space that are covered by multiple geometries (i.e., overlaps), but placement of geometries in space can be highly skewed (e.g., numerous Greek islands are clustered in Cyclades, but there are only a few dispersed in the Northern Aegean). Hence, approximating spatial entities necessitates capturing the value domain of the underlying data distribution.

In this study, we concentrate on 2-dimensional rectangular shapes, which is a typical approach in spatial data indexing and query processing. In most cases, spatial objects are approximated by their minimum bounding rectangles (MBRs) that fully enclose their geometry in iso-oriented boxes with sides parallel to the axes. Then, these MBRs are hosted in a variety of spatial access methods (a.k.a. spatial indices) as indicators about the distribution and size of the exact geometries. In contrast to point access methods tailored for handing point data only (i.e., geometries with no spatial extent), spatial access methods can manage geometries with an extent, such as lines, polygons, or even polyhedra in d>2 dimensions.

A comprehensive survey of multi-dimensional access methods is available in [GG98]. A great variety of such data structures has been devised over the past decades, such as grids, k-d-trees [Ben75], quadtrees [Sam84], R-trees [Gut84], etc. Nowadays, many of them are integrated in leading geospatial DBMSs. For example, R-trees are used in Informix [Informix], MySQL [mySQL], Oracle Spatial [OracleDBMS], and PostgreSQL [PostgreSQL]; R-tree-over-GiST is included in PostGIS [PostGIS]; multi-level grids are employed in IBM DB2 [IBM-DB2] and MS SQL Server [MS-SQL-Server]; quadtrees were used in earlier editions of Oracle Spatial, etc. In addition, spatial indices are also built against RDF datasets, and several geospatially-enabled RDF stores incorporate R-trees, as we have verified in Deliverable D2.1.1 [GeoKnowD21] of this project.

Such spatial indices are most valuable when it comes to query processing against numerous spatial entities. Range queries are the most typical example of spatial searching. A range query needs to identify spatial entities that intersect with a given search region q, which may be an arbitrary polygon. In most cases, search region q is an iso-oriented rectangle with its faces parallel to the axes (hence, this type of query is also called window query [GG98]). Most systems perform query processing first against MBRs, using the spatial indices as much as possible. The result of a search query against any of these indices is a set of candidate answers, not the exactly qualifying geometries. These candidates may include false positives (due to approximate comparisons based on MBRs and not on exact coordinates), but no false negatives. For those candidates, their actual spatial extent (i.e., precise coordinates) has to be further probed for intersection with the search box q in order to provide the final (exact) answers. This is a typical case of the “filter-and-refinement” paradigm [RSV02], which is frequently used in spatial data processing.

2.1.2 Handling Spatial Predicates in Query Optimisation

Present-day DBMSs have native support for index-based selectivity estimation. For example, the spatial selectivity analysis for GiST indexes of PostGIS is integrated into the main selectivity module of PostgreSQL [PostGIS], whereas similar functionality for spatial index analysis, estimation and tuning exists in Oracle Spatial [OracleDBMS]. Such modules can provide high-performance execution plans for mixed queries that include both spatial and non-spatial (thematic) predicates. In fact, the majority of the submitted queries are mixed, because user requests often include thematic criteria and not just a
spatial area of interest. For instance, it is not expected that a user should generally ask for places in the city centre (i.e., the search region of her interest), but distinctively for hotels, restaurants, cinemas, museums, etc., occasionally specifying additional conditions as well (e.g., luxury hotels, or restaurants serving fish).

Query optimizers can take advantage of the already built spatial indices and the accumulated statistics on data distribution in order to determine cost-effective query execution plans. A suitable plan can intertwine evaluation of spatial and non-spatial operators in order to provide the answers faster. For example, if the spatial selectivity is low (i.e., only few geometries qualify), then it makes sense to evaluate the spatial predicate first using the spatial index, and handle all other non-spatial conditions afterwards. In case the spatial selectivity increases and more geometries are expected to satisfy the spatial predicate, the optimizer may choose different query plans. This way, optimizers can avoid full table scans, which may be very costly when millions of geometries are hosted in the database.

Quite similarly, RDF optimizers tend to separate evaluation of the spatial predicates from the thematic conditions (i.e., the graph-pattern criteria) in SPARQL queries. Most usually [BK12], the query optimizer has two alternatives: (a) execute the spatial part first and then match its results to the graph pattern, or (b) execute the graph-pattern criteria first, and then check the qualifying features with the spatial condition. The decision is based on which path will likely provide the fewest result bindings. With the exception of an early RDF prototype [BNM10] that attempted integration of spatial processing into core evaluation for spatial selections and joins, most RDF stores follow one of the aforementioned naïve strategies. As discussed in [BNM10], the query planner must be deeply modified to recognize the spatial index and optimize the joins between the spatial features it returns and other intermediate results from graph-pattern evaluation. In addition, the spatial index must also provide the spatial features in a way they can be joined with the entities resulting from the semantic evaluation, i.e., sorted according to unique feature identifiers and not on the basis of spatial proximity.

Of the available geospatially-enabled RDF stores, uSeekM [uSeekM] always starts by evaluating the spatial predicate in the underlying PostGIS repository and then proceeds to probe the graph patterns against its native Sesame store. Besides, Parliament [Parliament] gives precedence to the non-spatial part and afterwards evaluates the spatial predicates exhaustively on the intermediate results. Practically, non-spatial (thematic) and spatial selectivities of a query are ignored. In contrast, Strabon [GKK13] takes advantage of the integrated PostgreSQL/PostGIS optimizer and tends to select good query plans, thus offering better response times for mixed queries. Virtuoso’s optimizer supports geometry cardinality sampling and selects good execution plans for combinations of spatial and thematic joins [Virtuoso].

2.1.3 Our Approach to Spatial Selectivity Estimates from RDF Datasets

Our goal in this study is to identify cost-effective, fast, and reliable means of estimating the selectivity of spatial predicates against large RDF datasets with millions of geometric objects. The rule is that, in presence of indices, a spatial query can be optimized such that its most selective part (spatial or non-spatial) is executed first. Thus, we examine how spatial selectivities can be efficiently estimated on-the-fly, so that they can be utilized by the RDF query optimizers when choosing among several query plans. With accurate predicate selectivity values, an appropriate join order would be facilitated, whereas less intermediate results would need to be retained during query evaluation.

Since having accurate spatial selectivities pays off in case of mixed queries, we propose variants of state-of-the-art spatial indexing methods. Most geospatially-enabled RDF stores make use of spatial indices (like R-trees), so enhancing them with functionality for selectivity estimation would certainly
be affordable. What we actually suggest is to turn a given spatial access method into a spatial histogram. Thus, the 2-d space can be approximated as a 2-dimensional histogram whose buckets are associated to the spatial density of the corresponding regions, i.e., the number of objects which overlap that area. More specifically, we have implemented variants for the following spatial access methods:

- Uniform grid partitioning;
- Quadtrees; and
- R-trees.

All derived histograms take into consideration the spatial density and spatial skew of the geometries in a concise manner, so that they can yield accurate selectivity estimates as fast as possible. The usual functionality of the index remains intact, so it can support spatial query evaluation.

As explained in [APR99], the problem of constructing histograms for spatial data can be resolved by using equi-area or equi-count heuristics. The input space is subdivided into buckets (or regions) which have the same area or the same number of data objects, respectively. But when applied against datasets containing complex geometries with large spatial extent, such techniques may result in histograms which provide poor selectivity estimates, because their buckets tend to have large MBRs. Besides, some buckets may have too few objects, while others may contain a larger number of entities, hence affecting the final selectivity estimates. Still, there are two issues that must be tackled for spatial histograms [BRHA13]:

a) Geometries should be grouped into buckets based on spatial proximity. As discussed next, there is no common rule on this, as space-driven techniques (like grid partitioning or quadtrees) exist alongside data-driven structures (like the family of R-trees). As a result, buckets may have overlapping contents or may be completely disjoint.

b) Selectivity should be estimated directly from the resulting buckets, without accessing the detailed geometries. If the buckets are disjoint, then this step is straightforward. In previous proposals [APR99, BRHA13], each bucket reports the number of intersecting geometries as well as the average height and width of the respective MBRs. These values can then be used to determine the predicate selectivity value for different query regions. In our evaluation, we follow a simpler methodology, which properly traverses the top tiers of the spatial index and only examines the object counts per bucket.

In the following sections, we present the variants of several state-of-the-art spatial indexing methods intended for use as spatial histograms in RDF spatial query optimisation.

2.2 Grid-based Histograms

2.2.1 Overview of Uniform Grid Partitioning

A regular decomposition of a given area of interest is a space-driven access method [Jon97]. A simple, yet quite effective spatial decomposition scheme is to simply overlay the space with a **uniform grid** of equi-sized cells (also called **buckets**). Its **granularity c** controls the number of cells along each axis, leading into a total $c \times c$ cells when subdividing the 2-d space. Given grid uniformity, converting (i.e., hashing) a spatial coordinate into the corresponding cell is trivial. Hence, each geometry is assigned to those cells it overlaps with. Thanks to its simplicity, grid partitioning allows fast indexing of geometries and is widely used especially for point objects or dynamically changing geometries in main memory (e.g., moving objects [SSC+09]). Additionally, given a particular cell, neighbouring cells
can be easily identified (e.g., for nearest-neighbour search).

This spatial index is far simpler to maintain than other data structures (e.g., R-trees, discussed next), since it only involves lists of objects per cell. However, the number of features may vary widely among cells; some may be completely empty, whereas others may contain too many features. In the latter case, this may incur delays when probing objects in the refinement stage of query evaluation [RSV02].

Nonetheless, choosing a suitable cell size is a very subtle issue. In particular, performance may be negatively affected in several cases:

- If grid granularity $c$ is too fine and a large object is inserted or moved, then a large number of cells must be updated.
- If the grid is too coarse (i.e., small $c$), then each cell is likely to hold a large number of objects (i.e., locally high density). This may incur an increased cost for searching, as each of these objects may need comparison with the search area.
- If the sizes of the indexed objects vary significantly (i.e., there are many small and many large geometries), then neither a fine nor a coarse grid would be adequate.

As a rule of thumb, cell size is often selected to be large enough to fit the largest object at any rotation (i.e., no more than four cells would be needed for such an object in a 2-d grid). However, a suitable choice for granularity $c$ actually depends on the data at hand, so it should be chosen empirically after testing several $c$ values against the particular dataset (as further discussed in our evaluation in Section 2.5).

Grid partitioning can greatly assist in spatial searching, especially for range queries that want to identify objects intersecting a given rectangular range $q$. When such a spatial predicate must be checked against the grid, this just requires examination of the cells overlapping with the area $q$. In case that a cell falls completely within $q$, all its geometries are returned as answers. But this is not the case for cells partially covered by the query area $q$; each object assigned to any such cell must be probed separately against $q$, because it may not intersect $q$ at all. A typical decomposition into 5x5 cells for several types of geometries (points, lines, and polygons) is illustrated in Figure 1. When issuing the range query depicted with the green rectangle, nine cells are involved. One of them is inside the range and hence all its objects qualify; the other eight require further examination in order to identify qualifying geometries.

### 2.2.2 Employing Grid-based Histograms for Spatial Selectivity Estimation

In order to use grid overlays in selectivity estimation for spatial predicates, we propose that each square cell be enhanced with a counter, which simply counts the geometries assigned to that cell. This leads to a 2-dimensional analogue of a histogram, which we call **Cell Histogram**. Once a query range $q$ is given at run time, the corresponding counts of its overlapping cells can be summed up to provide a fair estimation of the geometries involved in the search. The encircled values at the corner of each cell in Figure 1 indicate these counts for the given setting.
Figure 1: Example of a Cell Histogram based on 5x5 grid decomposition

This approach poses no problem for point data, as each such location is always referred to by a single cell, so each point is counted only once. Unlike point objects which either fall inside a cell or not, a line or polygon geometry may span several cells. In case that the MBR of such a geometry $g$ overlaps with multiple cells, $g$ must be referred to by each of these cells; thus a single geometry may be incrementing the counter of multiple cells. In the example setting of Figure 1, the MBR of object $o_1$ affects the counters in four cells. In case of a spatial query that covers all those four cells, this single object will be counted four times, incurring a 300% overestimation. Obviously, when estimating selectivity from cell histograms, there is no way to avoid this multiple-count problem, other than with a detailed examination, which of course has a prohibitive cost during query plan selection. Indeed, when a range query $q$ (shown with a green rectangle) is examined against the cell histogram in Figure 1, the estimated selectivity from the six overlapping cells is 26. Obviously, this deviates too much from the exact number of 15 geometries that query $q$ actually intersects.

An alternative policy is to count each geometry only once, no matter if it covers multiple cells in the underlying grid. Incrementing the counter of the cell that contains the centroid of the geometry is a plausible option; hence, we call this scheme a Centroid Histogram. In this case, the sum of the count values per cell is equal to the number of geometries indexed by the grid. Figure 2 illustrates the values of cell counters according to this approach, which provides an estimated selectivity of 22 for the range query shown in green colour. However, this histogram has another serious drawback: it does not account for geometries that partially overlap with a cell, so the accuracy of spatial selectivity estimates may be significantly biased.
Figure 2: Example of a Centroid Histogram based on 5x5 grid decomposition

**Euler Histograms** were introduced in [BT98] to specifically address the multiple-count problem, owing their concept to the well known Euler's Formula in graph theory. Compared to the conventional grid partitioning, an Euler histogram allocates counters not only for grid cells, but also for cell edges and vertices. Figure 3 depicts the Euler histogram for the same geometries as the example setting as in Figure 1. Note that the original geometries have been omitted for clarity. Having additional counters for grid cell edges (shown in black colour) and vertices (shown in red), it is possible to distinguish between a large object spanning several cells and several small objects fully contained within a single cell.
Figure 3: Example of an Euler Histogram based on 5x5 grid decomposition

As analyzed in [BT98], given a \(d\)-dimensional range query \(q\), its selectivity can be calculated with an Euler histogram with the following formula:

\[
\text{Selectivity}(q) = \sum_{0 \leq k \leq d} (-1)^k F_k(q)
\]

where \(F_k(q)\) is a \(k\)-dimensional facet intersecting query \(q\). In particular, a 0-dimensional facet is a vertex, a 1-dimensional facet is an edge and a 2-dimensional facet is a cell. Clearly, this is close to the Euler’s formula for planar, embedded, connected graphs [BKOO00]. If the query range aligns with the grid (i.e., no partial overlaps exist with any cells), then the estimate given by this formula is accurate and incurs no error at all.

For example, the query shown with a green rectangle in Figure 3, intersects 4 vertices (0-d facets), 12 edges (1-d facets), and 9 cells (2-d facets) of the Euler histogram. Summing up the counts for each particular group of facets according to the formula, gives a selectivity estimation of

\[-1^0 \times (0+0+0+0) + (-1)^1 \times (1+0+0+1+0+0+0+0+0+1) + (-1)^2 \times (4+3+3+2+4+8+0+1+3) = 25,
\]

which is well above the number of 15 geometries that really intersect the query range.

As our empirical validation in Section 2.5 indicates, grid-based histograms can provide selectivity estimates very fast (especially for coarser granularities), but these values largely deviate from the exact number of qualifying objects.

2.3 Quadtree-based Histograms

2.3.1 Overview of Standard Quadtrees

The term quadtree (also known as octtree in 3D-spaces) usually refers to a class of hierarchical data structures based on the principle of recursive decomposition of the 2-dimensional space. As discussed in [Sam84], they can be differentiated according to:

a) the type of data to be represented, because quadtrees can be used for point data, polygons, curves, surfaces, and volumes;

b) the decomposition rule that controls the subdivision of the universe, either into equal parts on each level (i.e., regular polygons like triangles, squares or hexagons) which is termed a regular decomposition, or guided by the input; and

c) the resolution (variable or not).

The most typical variant is known as the region quadtree [FB74], which is based on successive subdivision of the area of interest into four equal-sized, disjoint quadrants. Although the term quadtree usually refers to the 2-dimensional variant, the basic idea applies to an arbitrary number \(d\) of dimensions. Like the \(k\)-d-tree [Ben75], the quadtree decomposes the universe by means of iso-oriented hyperplanes into adaptable buckets (i.e., its quadrants). Yet, quadtrees are not binary trees: for \(d\) dimensions, the internal (i.e., non-terminal) nodes of a quadtree have \(2^d\) descendants, each corresponding to an interval-shaped partition of the given subspace. These partitions do not have to be of equal size, although that is often the case. The root node always corresponds to the entire universe. For example, in case that \(d=2\), each internal node has four descendants, each corresponding to a rectangle. These rectangles are typically labelled counter-clockwise as its NE (northeast), NW (northwest), SW (southwest), and SE (southeast) quadrants of their parent node. The decomposition
into subspaces is usually continued until the number of objects in each partition is below a given threshold, known as the quadrant’s capacity $c$.

Once a quadrant obtains more than $n$ geometries, it splits into four children quadrants. Thus, quadtrees are not necessarily balanced; subtrees representing densely populated regions may be deeper than others. The leaf (i.e., terminal) nodes of the tree correspond to those blocks for which no further subdivision is necessary.

In more complex versions of the region quadtree, such as the PM quadtree [SW85], it is also possible to store polygonal data directly. This variant divides the quadtree regions (and the data objects in them) until they contain only a small number of polygon edges or vertices. These edges or vertices (which together form an exact description of the data objects) are then attached to the leaves of the tree.

Searching in a quadtree is very similar to searching in an ordinary binary search tree. At each level, a decision should be made on which of the four subtrees requires further search. In the case of a point query, typically only one subtree qualifies, whereas for range queries there are often several. This search step is recursively repeated until the leaves of the tree are reached, which eventually yield the candidate answers.

2.3.2 Using MBR-Quadtrees as Spatial Selectivity Estimators

In the case of spatial indexing that only involves point data, a standard region quadtree is very efficient. But for more composite geometries (e.g., curves, polygons), their MBRs should be indexed instead. It turns out that a region quadtree may index each MBR multiple times. Indeed, it may frequently happen that a given MBR overlaps with several quadrants, so it should be indexed in each one of them. This may be fine in terms of searching, but it would certainly lead to significant discrepancies in selectivity estimation, as a single object would be counted multiple times (i.e., once per each quadrant it belongs to).

Therefore, we propose a variant structure, which we call MBR-quadtree, as it is specifically intended for indexing MBRs and offering reliable selectivity estimations for geometries. This quadtree-like data structure is loosely inspired by the MX-CIF quadtree devised by [Ked82] for representing a large set of very small rectangles in VLSI design rule checking. It is also reminiscent of the notion of irregular octtrees introduced in [SP03], where each object is only held once within the tree, but some objects (those crossing a partitioning plane regardless of size) are retained at a higher position in the tree. Our goal is to rapidly identify which geometric objects intersect a given query range in user requests against high volumes of RDF spatial datasets.

The MBR-quadtree is organised in a similar way to the region quadtree, except that it stores entries in internal nodes as well. Our policy is to store each MBR in the quadtree node whose associated quadrant provides the tightest fit without intersecting the MBR. Once a node $A$ overflows because it contains more than $c$ MBRs, its region is subdivided into four equal-sized quadrants. Then, all MBRs in node $A$ are checked for inclusion in the newly formed children nodes (and subsequently deallocated from $A$). In case that an MBR is not fully contained within a child quadrant, it remains assigned into the same node $A$ and not considered member of any of its descendants. Thus, geometries (i.e., their MBRs) are associated with both terminal (leaves) and non-terminal (internal) nodes. This is an important difference with region quadrees, where all geometries are always contained in the leaves (although at various depths). Our partitioning policy also differs from the one in MX-CIF quadtree, in which a region is repeatedly subdivided into quadrants until blocks are obtained that do not contain
rectangles, whereas objects within a node are organised by means of binary trees. In our structure, subdivision into quadrants occurs once capacity $c$ is exceeded for leaves. In contrast, internal nodes may contain more than $c$ geometries, if necessary. Figure 4 illustrates the resulting subdivision of the space (left) for the input geometries when node capacity is $c=4$, as well as the counts retained in each node (right). Note that NE, NW, SW, SE quadrants of the corresponding MBR-quadtree are listed from left to right at all levels of the tree in Figure 4.

Figure 4: Example of a MBR-Quadtree and its associated histogram for quadrant capacity $c=4$

In addition to holding entries for MBRs, each node of the quadtree is augmented with two counters:

a) The number $n$ of local entries pertaining to that specific node. Note that an internal node can also store MBRs in case that they do not fit within any of its children. In Figure 4, this counter is shown as an encircled red value at the centre of the respective quadrant region (i.e., the common corner of its four children nodes). For example, the root node has $n=3$ as it holds three MBRs itself: two of them ($o_1$ and $o_2$) are crossing the partitioning line along the $x$-axis, whereas another one (for polygon $o_3$) is found along the partitioning line of the $y$-axis. Another MBR (for curve $o_4$) is also assigned to the SE quadrant below the root, since it cannot fit into any of its descendant nodes.

b) The total number $m$ of MBRs contained in its subtree. This value can be updated in a top-down fashion whenever a new MBR is inserted into the tree from the root. For every quadrant overlapping with this new MBR, its counter $m$ is incremented and the process is repeatedly applied until either the new MBR reaches a suitable leaf where it is fully contained or is assigned into an internal node (as explained before). In Figure 4, these $m$ values are shown close to the corner of each quadrant region. For example, the NE quadrant of the root fully contains $m=4$ geometries (three points and a curve), so it filled up to its capacity ($c=4$) and requires no further partitioning. Instead, its sibling NW quadrant is denser and includes $m=20$ geometries, hence it is recursively split even
When it comes to selectivity estimation for a given rectangular range query \( q \), MBR-quadtree can be traversed as a typical quadtree starting from the root node. If the quadrant corresponding to a node at a given level is fully contained within region \( q \), then visiting its subtree can be safely avoided, since all relevant selectivity is readily available from the two counters \( n \) and \( m \) in that particular node. In case that \( q \) intersects (but not fully covers) a given quadrant, searching proceeds to the next level by examining each of its subquadrants overlapping with \( q \). Count \( n \) of geometries locally held in that quadrant are added to the estimate, but value \( m \) is ignored because the subtree will be further explored and a more accurate estimate can be obtained. Of course, if a quadrant does not overlap with query \( q \), then it can be safely pruned. When estimating selectivities, leaf entries are never examined in detail. For the example query shown with a green outline in Figure 4, the estimate from this MBR-quadtree is that 19 geometries may be intersecting \( q \). This value is obtained from the nodes shown with a green shadow in Figure 4 and incorporates the values from local counters in the root and its SE quadrant. This estimate is closer to the exact number of 15 qualifying geometries compared to the selectivity obtained from grid-based histograms presented in Section 2.2. As our experimental evaluation in Section 2.5 verifies, this strategy also keeps the estimation cost very low, since typical range queries require probing of just a few quadrants in order to get a fair estimation of the selectivity.

2.4 Data-driven Spatial Histograms

2.4.1 Overview of the R-Tree Family

An R-tree [Gut84] organises a hierarchy of nested \( d \)-dimensional rectangles. Each node of the R-tree corresponds to a disk page and represents a \( d \)-dimensional MBR. For any internal node, all its descendants are contained within its MBR. However, MBRs of nodes at the same tree level may overlap. For a leaf node, its MBR is the \( d \)-dimensional minimum bounding rectangle covering all geometries indexed by this leaf. Leaves do not store the full description of each geometry (i.e., its detailed coordinates), but only its MBR and a reference to that description on disk. The R-tree has the following properties:

- Every node contains between \( m \) and \( M \) entries, unless it is the root. The lower bound \( m \) prevents tree degeneration and ensures efficient storage utilization. Once a node has less than \( m \) descendants, that node is deleted and its children are distributed among its sibling nodes (tree condensation). The upper bound \( M \) signifies the maximum capacity in number of entries for each node. \( M \) is derived from the fact that each tree node corresponds to exactly one disk page.
- The root node has at least two entries, unless it is a leaf.
- All leaves are at the same level, so R-tree is height-balanced and its height is logarithmic to the amount of indexed geometries.

To insert a given geometry \( g \), its corresponding MBR(\( g \)) and an object reference must be inserted into a leaf. Only a single path is traversed from the root to a suitable leaf. At each level, the child node whose corresponding MBR needs the least enlargement to enclose MBR(\( g \)) is chosen. Therefore, \( g \) is inserted only once and not dispersed over several buckets. At the leaf level, the chosen node is adjusted to accommodate the new entry MBR(\( g \)), by enlarging the MBR of that leaf (if necessary) and
propagating the change upwards. If the capacity M of that leaf is exceeded, then that node is split, and its entries are distributed between the old and the new page. The MBRs of both leaves are adjusted accordingly, and the split may propagate up in the tree. For node splitting, [Gut84] suggested several strategies, including a simpler one with linear time complexity and a more sophisticated one with quadratic complexity.

Besides, deleting an entry requires its removal from the leaf where it resides, followed by an adjustment of MBRs upwards, which may possibly affect even the root node. In case that a node occupies less than m entries, they are all removed from the tree and reinserted afterwards, which also involves adjustment of any affected MBRs.

In terms of searching, the R-tree behaves similarly to the B-tree. At each node A, all its entries are tested for intersection against the search box $q$. Then, all qualifying entries are visited and this pattern is repeated until the leaf level or until no further box overlaps are found. In contrast to the insertion policy, searching may need to explore several paths of the R-tree, since multiple sibling entries could be overlapping with box $q$. In the worst case, every index page must be visited. This step may involve additional disk accesses and considerable computation.

In the original R-tree [Gut84], various policies are examined towards minimizing the overlap during insertion. Later work led to more sophisticated policies and R-tree variants. For example, the packed R-tree [RL85], computes an optimal partitioning of the universe and a corresponding minimal R-tree, but requires all data to be known a priori. The $R^+$-tree [SRF87] is based on clipping, so there is no overlap between MBRs at the same tree level; objects that intersect multiple MBRs must be stored in several different pages. The Hilbert R-tree [KF94] combines the overlapping regions technique with space-filling curves.

However, the insertion phase in R-tree is most critical for good search performance. Therefore, the $R^*$-tree [BKSS90] introduces a "forced reinsertion" policy; deletion and searching remain practically identical to those in the original R-tree. In $R^*$-trees, if a node overflows, it is not split right away, but a portion of its entries are removed and reinserted into the tree. It is suggested that about 30% of the maximal number of entries per page should be reinserted. In addition, an improved node-splitting policy is proposed, which is based on the plane-sweep paradigm [PS85] with the following goals:

- Overlap between MBRs at the same tree level should be minimized. The less the overlap, the smaller the probability of traversing multiple search paths.
- Perimeters of MBRs should be minimized. This rule tends to create square-like MBRs, which are the most compact rectangular representation.
- Storage utilization should be maximized.

In tests against a variety of data distributions [BKSS90], the $R^*$-tree was able to provide up to 50% performance gain when compared to the R-tree, mainly thanks to its insertion and node-splitting policies. Storage utilization is also benefited from this method, but at the expense of higher insertion costs. Overall, the $R^*$-tree is considered the best spatial index among all members of the R-tree family.

2.4.2 Using Aggregate R-trees as Spatial Selectivity Estimators

As we have verified in our survey for Deliverable D2.1.1 [GeoKnowD21], R-trees and $R^*$-trees are being extensively used for indexing geometries in RDF datasets. Such an index greatly facilitates query answering for various types of queries, but does not provide any assistance in selecting a suitable query plan. In order to remedy this deficiency, we propose to enhance every node in the tree with a
counter, which simply reveals the number of geometries indexed by its subtree. It is no surprise that such a variant R-tree has already been utilized for OLAP operations over spatial features.

**Figure 5:** Example of an R-tree with at least \( m=2 \) and at most \( M=5 \) per node

Indeed, the aggregate R-tree (\( aR \)-tree) [PKZT01] improves the original R-tree [Gut84] towards aggregate processing. More specifically, in each intermediate entry, \( aR \)-tree stores summarized data about objects residing in its underlying subtree. In the particular case of count aggregates, each MBR also retains the number of objects therein (i.e., indexed by its subtree). Note that double-counting of geometries never occurs, since each MBR belongs to a single cluster only.

As illustrated in Figure 5, the \( aR \)-tree is built following the same principle of clustering over the spatial objects; hence, its structure is similar to that of an R-tree. The blue rectangles indicate MBRs at the bottom level; the red rectangles represent their parent MBRs. In this simple case, the tree has just two levels as shown in Figure 6, so the red rectangles belong to the root. The numbers in each node indicate the amount of geometries indexed by each MBR.

As soon as a spatial predicate \( q \) (e.g., the green box in Figure 5) is applied against this index, it would be feasible to provide an approximate, yet very fair estimation of the affected geometries. Like a typical range query against the R-tree, it is simply a check against the MBRs that fully or partially
overlap with the query box. The algorithm should descend the tree (perhaps at multiple paths) to
detect such MBRs and simply sum up their counts. In the example of Figure 6, the algorithm will start
searching from root entries \( R_2, R_3, \) and \( R_4 \) which are the ones that overlap with the query range \( q \).
From their respective subtrees, it is found that entries \( f, g, k, l, \) and \( m \) actually intersect range \( q \). The
counters maintained in these five nodes will finally provide an estimated selectivity of 18, which is
very close to the exact number 15 of qualifying geometries and much better than all other spatial
indices. However, such accuracy comes at a cost; building any index of the R-tree family takes more
time and its reorganisation suffers in case of frequent data updates. Moreover, sometimes the number
of tree nodes that need inspection may be high, as we discuss in our evaluation study in Section 2.5.

2.5 Experimental Evaluation

2.5.1 Use Case

In our use case scenario, OpenStreetMap [OSM] data for Great Britain covering England, Scotland,
and Wales as of 05/03/2013, were downloaded in ESRI shapefile format [OSM_GB]. Of the available
OSM layers, only those concerning road network (roads), points of interest (points) and natural parks
and waterbodies (natural) were actually utilized (Figure 7). These layers were chosen as
representatives for each geometry type (respectively points, linestrings, and polygons), and they also
contained the larger number of features compared to other similar layers. From each original layer, we
actually utilized the geometry, keeping also the unique OSM identifier as a reference, as detailed in
Table 1.

All data was used “as is”, without modifying the original geometric information. Features were
already georeferenced in WGS84, so no transformation was necessary. Original shapefiles were
converted into RDF triples with TripleGeo, an open-source utility [TripleGeo] developed earlier in
this project specifically for such tasks. Details about this tool can be found in Deliverable D2.2.1
[GeoKnow22].

<table>
<thead>
<tr>
<th>OSM Layer</th>
<th>Geometry type</th>
<th>Description</th>
<th>Attributes used</th>
<th>File size (MBytes)</th>
<th>Number of features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>Point</td>
<td>Points of general interest (bus stop, pub, hotel, traffic signal, place of worship, etc.)</td>
<td>shape, osm_id</td>
<td>74.3</td>
<td>590390</td>
</tr>
<tr>
<td>Roads</td>
<td>LineString (spaghetti)</td>
<td>Road network characterized according to OSM classification (motorway, trunk, primary, secondary, tertiary, residential etc.)</td>
<td>shape, osm_id</td>
<td>706</td>
<td>260140</td>
</tr>
<tr>
<td>Natural</td>
<td>Polygon</td>
<td>Surfaces like parks, forests, waterbodies (lakes, riverbanks etc.)</td>
<td>shape, osm_id</td>
<td>152</td>
<td>264570</td>
</tr>
</tbody>
</table>
2.5.2 Experimental Setup

For conducting experiments, we have set up a XEN hypervisor on an evaluation Linux server (Intel Core i7-3820 CPU at 3.60GHz with 10240KB cache), which is capable of hosting a group of Virtual Machines (guest VMs). Each VM used in these experiments was given 16GB RAM, 2GB swap space, 4 (virtual) CPU cores and 40GB disk of storage space.

Since our focus in this evaluation is strictly on selectivities of spatial predicates, all algorithms were implemented in C++ as standalone classes without any interaction with database systems or RDF stores. Apart from the great difficulty of coupling our techniques with the intricate query execution engine employed by such systems, it would not provide any practical advantages towards assessment of the methods. All implemented data structures reside in main memory and can be used to index MBRs of typical 2-d geometries (points, lines, and polygons). Geometries are parsed from WKT serialisations in an input RDF file, and their MBR gets calculated. Afterwards, only this MBR is used in indexing and query evaluation.

We have also generated query workloads for testing the spatial selectivity estimators of each method. In order for them to be representative and conform to the actual data distribution, we have randomly taken point geometries from the dataset and used them as centres of the respective query ranges. In total, 10000 points were used, so that an equal number of query rectangles could be created. As the size of a query range has a strong influence on selectivity, we generated several workloads with differing range size (but always having the centre points obtained before). Spatial range of queries is expressed as percentage (\(a\%\)) of the area of the entire spatial Universe \(U\) (i.e., the iso-oriented MBR that encloses Great Britain).

However, ranges are not necessarily squares with the given centroid, because we randomly modify their width and height in order to get arbitrarily elongated rectangles of equal size \(a\). No semantic criteria (e.g., name, type) are involved in the queries, since index structures handle geometric information only and our study focuses particularly on spatial selectivity estimation. Figure 7 depicts

![Figure 7: Data Universe and indicative query workload used in the evaluation study](image)
the query rectangles with size $a=1\%$ of the Universe. It can be verified that most queries involve large metropolitan areas mainly in the south (London) and the Midlands (Birmingham, Manchester, Leeds, etc.), since the density of input geometries is much higher there. In Table 2, we summarize experimentation parameters and their respective ranges; in bold is the default value used in most diagrams, unless otherwise specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input geometries</td>
<td>345600</td>
</tr>
<tr>
<td>Number of range queries</td>
<td>10000</td>
</tr>
<tr>
<td>Range size ($a%$ Universe)</td>
<td>0.1, 0.5, 1, 2, 5, 10</td>
</tr>
<tr>
<td>Grid granularity $c$ (per axis)</td>
<td>100, 200, 300, 400, 500, 1000</td>
</tr>
<tr>
<td>Quadrant capacity (for quadtrees)</td>
<td>25, 50, 100, 200, 400</td>
</tr>
<tr>
<td>R-tree page size (in KBytes)</td>
<td>1, 2, 4, 8</td>
</tr>
</tbody>
</table>

In comparing the various techniques, we used the following metrics:

a) **Preprocessing Cost**: This is the total time (in seconds) taken by the construction algorithm for each technique to preprocess the entire input RDF data. This includes the cost of calculating MBRs for all geometries identified via their WKT serialisations, building appropriate data structures, and generating the buckets that will be used in selectivity estimation for queries.

b) **Average Number of Buckets Inspected per Query**: This represents the average number of buckets that each query has to visit in order to get an estimation of its selectivity. In grid partitioning schemes, this is practically the average number of cells overlapping with query rectangles. For quadtrees and R-trees, it is the average number of internal nodes (not leaves) with quadrants or boxes overlapping with the query ranges.

c) **Average Estimation Cost per Query**: Given a spatial histogram, this value (in milliseconds) indicates the average time it takes for a query in order to estimate its selectivity. Obviously, this cost depends on the number of buckets that need be probed.

d) **Average Relative Error**: This is the ratio of the error in an estimate to the actual size of the query result averaged over the entire query workload. Estimates are based on a spatial filtering operation which selects all geometry MBRs intersecting the query rectangle. More specifically, if $SE_i$ is the estimated answer according to the histogram and $AS_i$ is the actual number of entities qualifying for a given range query $q_i$ then the average relative error $RE$ for the entire query workload $Q$ can be computed as:

$$RE = \frac{\sum_{q_i \in Q} |AS_i - SE_i|}{\sum_{q_i \in Q} AS_i}$$

2.5.3 Experimental Results

First, we examine the preprocessing cost required by each histogram scheme. Then, we study the performance of each separate type of histograms for the same query workload in order to identify suitable parameterizations. Finally, after choosing suitable settings per histogram type, we proceed to compare their performance and quality in selectivity estimation for various query workloads.

**Preprocessing Cost.** As illustrated in Figure 8a, the time taken to build a uniform grid and then
perform its buckets, does not vary dramatically for the examined grid-based histograms and, in the worst case, it does not exceed 8 seconds. Of course, Centroid histograms incur the less preprocessing, as they only need to check where to assign a single point (i.e., the centroid of each MBR), which is absolutely trivial. Next, the cost for building Cell histograms is slightly higher, as each MBR may get assigned to multiple cells, thus several buckets need updating. Not surprisingly, creation of Euler histograms requires even more time, almost double as much as their Centroid counterparts. The reason is certainly the fact that each MBR may affect not only cell boxes, but cell edges and vertices as well, hence the increased cost. With respect to a varying granularity of the grid partitioning, we observe no significant fluctuation in the preprocessing cost for Centroid and Cell histograms. However, the cost for Euler histograms is rising linearly with the grid granularity, because more cell edges and vertices are involved. Indicatively, an Euler histogram with 100x100 cells results into 20200 edges and 102001 vertices, whereas for 1000x1000 decomposition, buckets for 2002000 edges and 1002001 vertices must be maintained.

Regarding quadtree-based histograms, Figure 8b indicates that the cost gets slightly reduced for increasing values \( c \) in quadrant capacity. This is natural, because the more geometries accommodated in a quadrant, then less the nodes that will be created, so the resulting quadtree will have less depth. For the entire dataset, if each quadrant can hold up to \( c=50 \) geometries, there are 194405 nodes in 20 levels. In case that \( c=400 \), then only 27693 nodes are employed, organised in 18 levels. Overall, building a quadtree histogram for the entire input does not take more than 5 seconds in the worst case, which is quite impressive.

Compared to the aforementioned approaches, preprocessing cost for building aR-tree histograms is much more expensive, and it escalates with increasing page sizes for the nodes, as depicted in Figure 8c. The cost is increased primarily because of the complex logic behind node splitting, especially for the aR*-tree variant, which additionally applies forced reinsertions of already indexed geometries. Performing clustering against MBRs is costly, particularly when nodes include more entries. For a page size of 1 Kbyte, a node can hold between \( m=8 \) and \( M=21 \) entries and the R-tree has 6 levels. When the page size is 8 Kbytes, the R-tree has only 4 levels, since each node indexes between \( m=68 \) and \( M=170 \) entries. In the latter case, whenever a node has to split, more comparisons must be made among its indexed MBRs. On the whole, data-driven spatial histograms incur a cost of more than an order of magnitude higher compared to their space-driven counterparts. But, we stress that this preprocessing cost is paid only once, when the histogram is built. As long as there are no frequent data updates that may cause reorganisation of the R-trees, there is nothing wrong in employing this type of histograms for selectivity estimation at query time.

![Figure 8](image.png)

**Figure 8:** Preprocessing cost of spatial histograms against the entire dataset
**Assessment of grid-based histograms.** Regarding grid-based histograms, Figure 9 illustrates the average performance and accuracy per query after testing against a workload with 10,000 rectangles ranging over \(a=1\%\) of the Universe. Clearly, more cells must be checked for finer granularities (the same number in all three variants), which also explains why the cost of getting a selectivity estimation escalates as well (Figure 9b). Euler histograms incur slightly more cost to provide their estimate, because they must probe buckets for cell edges and vertices as well. However, in every setting, the time needed to give a spatial selectivity per query is never more than 7 milliseconds. In terms of accuracy, Figure 9c reveals that Centroid histograms are consistently better than the other two variants for varying grid granularities. It is no wonder that the estimated selectivity gets far better when the cell size becomes smaller and smaller; then, there is less probability that the query range could be partially overlapping with large cells that contribute their counts and distort the final result. Euler histograms incur less than 20% error for various subdivisions, but they offer better estimates than Cell histograms. The latter get very unreliable for finer subdivisions, because of the inherent multiple-count problem. As more cells are employed, there is more chance that an MBR could be assigned to several cells and thus lead to overestimations.

![Figure 9](image1)

**Figure 9: Performance and accuracy of grid-based histograms**

**Assessment of quadtree-based histograms.** Checking against the same workload of queries with ranges over \(a=0.1\%\) of the entire area, we found that MBR-quadtrees are more competent into providing selectivity estimates both accurately and quickly. Indeed, the average number of buckets that each query has to check is lower (less than 800 buckets per query) and drops significantly for increasing quadrant capacities as shown in Figure 10a. As a result of this, the estimation cost per query (Figure 10b) gets reduced as well, and does not exceed 0.1 milliseconds in most cases.

![Figure 10](image2)

**Figure 10: Performance and accuracy of quadtree-based histograms**
As far as the estimation error is concerned, it can be seen from Figure 10c that it remains stable around 15%, no matter the quadrant capacity for this particular workload. The reason for this lies in the construction logic of MBR-quadtrees: since internal nodes also hold geometries, there is no double-counting in any node, hence all buckets examined contribute disjoint sets of potentially qualifying objects. This is a significant indication that the underlying histogram is robust enough in order to adjust even with skewed distributions of spatial data.

**Assessment of data-driven spatial histograms.** As shown in Figure 11a, several thousands of buckets in aR-tree histograms must be checked in order to provide selectivity estimates for range queries. This number drops when nodes can host more entries (thanks to increased page size), and it is much less for the aR*-tree variants. In fact, R*-trees perform better, as they inherently incur less overlap among MBRs of sibling nodes and thus excel in pruning power. This is also reflected in Figure 11b, which indicates a similar trend on the estimation cost per query, which never exceeds 3 milliseconds. But aR-tree histograms seem unrivalled when it comes to accuracy of the given selectivity estimates. Figure 11c verifies that the error is always less than 10% for standard aR-trees, reaching maximal discrepancy for large page sizes (8 KB). As for optimised aR*-trees, error never gets more than 5%, beating all other spatial histograms examined in our study.

![Figure 11: Performance and accuracy of data-driven spatial histograms](image)

**Comparison against varying range sizes.** For this set of experiments, we examine how each type of spatial histogram copes with varying range areas (all expressed as a% of the Universe). We show indicative results for specific parameterizations of each data structure, which have shown good behaviour in the previously discussed tests. In particular:

a) For grid-based histograms, we use a partitioning into 200x200 cells;

b) MBR-quadtrees have quadrants with maximum capacity c=50 geometries; and

c) Nodes of aR-trees and aR*-trees are given a page size of 2 KBytes.

Note that these settings do not generate the same number of buckets per histogram type. This is not possible to achieve, because the number of buckets for quadtrees and aR-trees is determined on-the-fly, according to the data distribution (also influenced by the insertion order in the case of aR-trees).
Regarding the number of buckets accessed per query, Figure 12a shows that it increases with greater range areas. Quadtrees consistently require the fewest node inspections, whereas standard aR-tree histograms must probe thousands of nodes to get a fair estimate. The situation is similar with respect to estimation cost per query, as plotted in Figure 12b. However, the cost in aR*-trees for larger-sized ranges is comparable to that of grid-based histograms, and always remains less than 1 millisecond per query. Note that Euler histograms incur increasingly more cost for larger query areas, as they have to collect partial counts for many more cells, edges and vertices of the grid. But what matters most is the accuracy of the estimation that each histogram provides. Figure 12c reveals that aR*-trees are always the best, as they incur the less error, even for smaller range sizes. Especially for smaller regions, it should be expected that the number of qualifying geometries may be significantly lower; hence, occasional false positives would lead to increasing errors. But aR*-trees never exceed a 5% error, whereas aR-tree estimates may be biased even by 20% for areas as small as a=0.1% of the Universe. However, Euler and Cell histograms tend to fail for either too small or too large range sizes, reaching an error up to 40%. Certainly, this originates from the choice of grid granularity, which is not flexible to cope with widely varying range extents. Unfortunately, one cell size cannot fit every arbitrary variation of query ranges, and this is an inherent drawback of flat, space-driven decompositions. But here, this setback is more pronounced because of the multiple-count problem in the associated buckets, which causes such wide discrepancies between the estimated and the actually qualifying geometries. Centroid histograms avoid this trap, so they can offer moderately good estimates, especially for larger ranges. In between, quadtree-based histograms offer fair accuracy in their estimates, with an error around 15% in most situations. So, taking into account the small cost of their computation (Figure 12b), quadtrees may be considered as a good trade-off between cost and quality.

**Visualisation of spatial histograms.** An indicative arrangement of the MBRs for input geometries in several spatial indexing schemes is illustrated in Figure 13. Each index has been built according to the default settings specified in Table 3. Note that the resulting visualisations are not shown at the same map scale, as each index leads into decompositions with differing spatial extents.

The resulting arrangements of MBRs intuitively explain why quadtrees and R*-trees tend to provide more accurate selectivity estimates. Obviously, the flat subdivision employed by uniform grid overlays (boxes are the same for Cell, Centroid, or Euler histograms in Figure 13a) is in stark contrast with the smart, targeted partitioning attained by quadtrees (Figure 13b). Indeed, the higher the density of geometries, the finer the resulting quadrants in that area. Data-driven structures of the R-
tree family attempt a clustering of the input MBRs. Figure 13c and Figure 13d portray the MBRs of internal nodes for the standard R-tree and the optimised R*-tree, respectively. MBRs belonging to the top level (i.e., the root node) are shown in red colour, their children in green, and those in lower levels in blue and magenta. As it can be seen in Figure 13c, an R-tree tends to create large boxes with significant pairwise overlaps between them, as opposed to an R*-tree (Figure 13d), which produces more compact boxes with minimal overlaps. This is additional evidence that histograms which adjust themselves to the spatial density of the underlying geometries can cope better with the accuracy demands of spatial query optimisation.
2.6 Outlook

As it turns out from our evaluation, several spatial histograms appear as suitable candidates for use in selectivity estimation. This observation refers particularly to those having a hierarchical structure, i.e., to aR*-trees, aR-trees, and MBR-quadtrees. As soon as such an enhanced index is built, it can rapidly provide fairly reliable estimates; fine-tuning of the tree parameterization can improve estimates a lot. In our tests, spatial selectivity estimates per query can be given within 1 millisecond, which is quite affordable. Of course, much less time would be required if a more powerful, fully-fledged server were available.

Using such spatial histograms can be valuable during optimisation of queries involving both spatial and thematic conditions against RDF datasets. Semantic query optimizers could actually leverage selectivities obtained for the spatial and non-spatial blocks of such mixed queries and come up with more advanced, cost-effective execution plans. Taking into account the latent skewness and irregular density (characteristics inherent in most spatial datasets), is something that could pay off when pruning the search space towards the query answers. As long as the computed spatial selectivity is close to the actual selectivity value, probability of choosing a bad plan for executing a SPARQL query is expectedly low.

Our analysis and experimentation on spatial selectivity estimation, opens up perspectives to explore further alternatives. The first one of them concerns the domain over which spatial selectivities are computed. In our evaluation, we exhaustively traverse the internal nodes of a spatial histogram in order to sum up the object counts retained per qualifying bucket. However, as we also retain the average size (i.e., width and height) or MBRs under each node, it would be possible to take them into account as well, using a more advanced formula for selectivity estimation, as in [APR99, BRHA13]. Another scenario would be to consider only nodes from a few top levels of the hierarchical histograms, in order to keep estimation cost as low as possible. Of course, the trade-off in that case is the quality of the returned estimate. Besides, there is an interesting by-product from histogram traversals carried out during selectivity estimations. No matter the query plan eventually chosen, the nodes containing relevant geometries would have been identified after such a traversal. Certainly, false positives may be contained therein, but there are no false misses; all answers are definitely among those indicated by the visited MBRs. If those nodes are cached, they can be readily available for subsequent probing in later stages of the plan or exact geometric calculations during query refinement. Last, but not least, a major challenge is to couple such a spatial histogram into the query optimisation module of an RDF store, like Virtuoso [Virtuoso]. This would provide concrete evidence about its impact on fast execution of complex SPARQL queries with spatial predicates.
3. Virtuoso GeoSpatial Query Optimisation Enhancements

3.1 Data Types and Structures

Virtuoso is a SQL and RDF Database, which prior to the GeoKnow project only supported "point" geometries in SQL and RDF and has now been extended to support many common geometry type as detailed below.

Virtuoso has extensive geospatial functionality following the SQL/MM standard, the geospatial functionality for RDF is a straight mapping of SQL/MM. GeoSPARQL support is planned for the near future.

Virtuoso supports a single geometry data type that is distinct from all other data types. The geometry data type has subtypes that correspond to the OGC\(^1\) data types.

Geometries are specified with one or more points. The points occupy a position in a space of between 2 and 4 dimensions.

All points in a single geometry have the same number of dimensions.

The two first dimensions (X and Y) are specially recognized and an index may be constructed over these. The two additional dimensions, Z and M may be used to represent for example elevation and one extra application attribute of a point. Geometries have a spatial reference id that specifies whether these are on a flat or spherical surface.

For example, X, Y, and Z describe a position and M can describe a distance to the next point in a linestring. Suppose a linestring describes places traversed by a wire. The M for each point may describe the actual distance to the next point, which may not correspond to the linear distance.

The variants of point that define a geometry may be:

1. X, Y
2. X, Y, Z
3. X, Y, M
4. X, Y, Z, M

\(^1\)http://www.opengeospatial.org
The data types now supported are as follows:

- Point
- Point list
- Ring
- Pointlist
- Box
- Linestring
- Multilinestring
- Polygon
- Multipolygon
- Collection
- Curve
- Closedcurve
- Curvepolygon
- Multicurve

The external syntax for these data types is given by Information Technology -- Database languages -- SQL Multimedia and Application Packages -- Part 3: Spatial -- **13249-3:2011**. The serialization format is also known as WKT (Well Known Text).

Additionally, Virtuoso has been extended to support the PostGIS EWKT extension, allowing a spatial reference ID in a serialization. In addition to WKT and EWKT, Virtuoso geometries can now be generated from shape files.

### 3.2 Geometry Index

Virtuoso supports an R tree index containing bounding boxes of geometries. The bounding box is defined with a lower left and upper right corner point, where the X and Y coordinates of the point are considered. Any other dimensions of the point are ignored for the bounding box.

An R tree may be associated with a SQL table and as a special case there is one R tree that contains the bounding boxes of all geometries that appear in the object position in the RDF quads table.

Additionally, more R trees indexing specific subsets of the geometries occurring as RDF object values can be defined. Managing these is up to application logic.

### 3.3 SQL and RDF use of Geometries

In a SQL only use case, a geometry is stored in a column of type ANY or LONG ANY. An optional R tree index may be defined on the column. If so, the table must have a unique integer key that is used for referring to the row of the geometry from the R tree. The integer key will typically be the primary key of the table but does not have to be this.

In an RDF use case, a geometry may occur as an object value in the O (object) column of the RDF_QUAD table. In this case, the value is not the geometry but rather a reference to the RDF_OBJ table that contains the geometry itself. The rationale is the same as that for not storing string literals in string form: These will be replicated on many indices of the quad table and the resulting space utilization is undesirable. Furthermore, if the same geometry occurs on multiple quads, there is only one entry in RDF_OBJ. Thus column values of RDF QUAD are guaranteed to remain short, i.e. no longer than the longest numeric or date value.

---

3.4 Reading a Virtuoso Query profile plan

A query plan is essentially a pipeline of operations that can be read from top to bottom. The operator above produces the input for the operator below. Nesting is indicated by braces. Operators enclosed in braces must completely process their input before continuing with operators after the section in braces. A case in point is aggregation which is demoted by for { ... } .... The section in braces generates the rows to be aggregated and one it is fully evaluated the execution proceeds with the operators after the fork. These will typically read the temporary table filled by the operators inside the fork.

Query variables are demoted with <$v... >. A reference to a query variable set by an operator that is not immediately the preceding one is demoted by <r $... > via ...>. The position of the variable in the query state follows the $ sign.

After this is given the symbolic name of the variable, e.g. a column name or the name of the function of which this is the return value.

If this is a reference to a query variable assigned earlier in the plan with one or more operators between the reference and the assignment, the reference contains a via ... section which lists the operators that are between the reference and the assignment.

The operators are identified by a unique number. Internally this is the location in the query state where the operator keeps the mapping between its rows of input and output.

With vectored execution, an operator may receive any number of rows of input which it may reorder at its convenience. For this it must record for each row of output which row of input this corresponds to. Thus when referencing a query variable set earlier in the plan one must take into account the reordering and addition/dropping of rows in affected by the operators between the reference and the initial assignment. This is the meaning of the via section in the <r $xx > notation.

The last part of the query variable reference is a two letter indication of the type. If the first letter is followed by n this means that the variable is non-null. The type letters are $ for any, r for IRI, i for integer, d for double, f for float, t for datetime, N for wide string, s for string.

If the plan is a profile, i.e. it is annotated with actual times and cardinalities from running the query, each operator is preceded by a passage that indicates its percentage of total CPU time of the plan, its fanout and the number of rows of input. The fanout is the number of rows of Output divided by the number of rows of input.

The principal operators in a plan are:

- **Cluster outer seq start** - Queries and subqueries begin with this operator. It assigns numbers to the sets of parameter bindings the query gets as input. An optional section also begins with this. An optional section is the table or derived table on the right hand side of a left outer join.

- **From** - accessing an index

- **Sort** - Group by, order by, distinct or hash join build side

- **Group by read** - accessing the results of a group by filled in by a previous sort operator.
Topoby - Accessing a previously filled top k order by temporary space

Select - At the end of a plan, send the rows of input to the client. If Subq select return the rows of input as rows of output of the enclosing subquery.

Subquery - Indicates a SQL derived table, i.e. select in a from clause. The input is the output of the previous node.

Cluster location fragment - A set of operators partitioned and shipped to a partition in a cluster. There are single part fragments and distributed fragments (DFG). A DFG is a query fragment which begins with a stage operator. The stage is a partitioning exchange operator which routes its rows of input to the next operator in the appropriate partition based on the partitioning columns of the row of input.

Outer seq end - This demotes the end of an optional section, i.e. the table/derived table on the right side of left outer join. This has the effect of making aligned vectors of all the vector query variables assigned in the optional section. When producing rows for which there was no match in the optional section, these will be set to null. References to these variables downstream in the plan will be to the shadowing variables, listed in the shadow clause following this.

Some operators such as index access (From) have a section called vector param casts. This is used when the vector that serves as parameter is not aligned or not of the type required by the operator.

For example, a vector of column values in an index lookup might not be of the type of the column but might be of a type castable to this. In any case, the pata,key values for index lookup are made into solid aligned, materialized vectors. After this there is a sort step that orders the input according to key order, so that matches are generated from left to right.

In the event of a cluster partitioned operation, i.e. stage node or non-DFG location fragment (QF), the first table source (From) after the stage or fragment serves to both align and cast all inputs needed downstream in the DFG or QF. The operator first partitions the rows of input. Each partition then gets solid, aligned vectors of input values which are directly suitable for input to the first index lookup (From).

We consider and example: The setup is a cluster of 4 server processes with each 8 slices of data. There is one table, ct_c, which is joined to itself.

profile ('select count (*) from ct_c a, ct_c b where a.k2 = b.k1 option (loop, order)');

This returns a result set in which is found the below. This is also in the ql_plan column of the sys_query_log view.

In the below excerpt some lines are omitted. Comments are inline.

time   1.3e-06% fanout         1   1 rows in
s# 98 cluster outer seq start, set no <V $40 set_ctr in>
save ctx:()
-- the section inside the fork is executed to the end before the section after the fork is started.
s# 161 Fork 42
{
wait time   0.00012% of exec real time, fanout   0
The below introduces the 2 stage DFG which does most of the work.

s# 108  { Cluster location fragment 51 0 unordered
Params: ()
Output: ()
-- First partitioning step. Each of the 32 slices gets the identical plan to execute, this is a flood for all is this first step does not specify a partitioning key value.
time 8e-05% fanout  0  0 rows in
s# 115 Stage 1: Params: (<V $112 set_ctr in>)
-- This is the initial table scan. The operator is executed 32 times, once for each slice of data.
time 0.33% fanout  3.1e+06  32 rows in
s# 121 from DB.DBA.ct_c by ct_c  1.7e+08 rows
Key ct_c ASC (<V $28 a.k2 in>)
[copies params]
vector param casts: <V $40 set_ctr in> <V $112 set_ctr in>
-- This is the second partitioning step, each of the rows of output of the previous is sent to the slice corresponding to K2.
time 31% fanout  0.9  2e+08 rows in
s# 131 Stage 2: Params: (<v $134 q_a.k2 S136 in>, <V $138 set_ctr in>)
-- This is the 2nd index lookup, by a.k2 = b.k1 There are two parameters given in the vector param cast section, the first is a.k2 and the second is the set number. This is constant since the query is run on a single row of input but if the query were run on multiple rows of input this would specify which row of input the a.k2 comes from so that this would be aggregated in the right place. Many aggregates can be produced in a single invocation with multiple rows of input.
time 68% fanout  0  2e+08 rows in
s# 144 from DB.DBA.ct_c by ct_c unq  1 rows
Key ct_c ASC ()
inlined k1 = <v $134 q_a.k2 S136 in>
vector param casts: <V $28 a.k2 in> <v $134 q_a.k2 S136 in>, <r $112 set_ctr via S121> <V $138 set_ctr in>
-- This is the aggregation.
After code:
  0: sum <V $43 count i> 1 set no <r $138 set_ctr via S144>
  5: BReturn 0
}
}
-- The below returns the aggregate to the client. The combining of aggregates from different slices is implicit in the for operator whenever this ends with a cluster location fragment.
time 5.4e-07% fanout  0  1 rows in
s# 164 Select (<V $43 count i>)
3.5 Query Optimization

Producing an efficient query plan with geospatial operators depends on correct cardinality estimation. If a query has a geometry literal or an expression that evaluates to a constant geometry known at query optimization time, the bounding box of the geometry is used in sampling the R tree.

A geo predicate, e.g. st_intersects may specify a precision. If this is specified, the bounding box of the geometry is expanded by the indicated precision. If the precision is known but the geometry is not known, as is the case in a geospatial join, the precision is used to estimate the selectivity of the join. The total count of geometries in the R tree is known and the total surface area covered by the R tree is known. The fraction of the circle of precision over the total area is used as an indication of join selectivity. This is not very precise but is better than nothing.

When the bounding box is known at optimization time, the sampling operation makes multiple semi-random lookups in the R tree. At each level of the R tree, a certain number of subtrees intersect with the bounding box. Each estimate is equal to the product of the intersecting bounding boxes across all levels of the tree. Each lookup chooses one path from the root to a matching leaf and calculates the product of the count of intersecting geometries. The number of samples is proportional to the number of non-leaf nodes containing a match of the bounding box. Each sample traverses the tree at a different ‘angle’, i.e. since the subtrees have a linear order in the tree, one takes the n/1000 th matching leaf, where n ranges from 0 to 1000 with a certain step. With 20 matching subtrees, n=0 takes the leftmost, n = 100 takes the 2nd leftmost etc.

The process is repeated until a maximum number of samples is taken or the standard deviation between all samples falls under a threshold.

With under 20 samples one usually gets an estimate that is within 30% of correct, where the match count is in the millions and the total count in the tens of millions.

Below are some examples of sampling against the LOD2 geobench data set. The numbers below are counts of geometries matching a st_intersects . The first number is the count of geometries estimate based on sampling and the second is the actual count:

<table>
<thead>
<tr>
<th>Sampled</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>38000</td>
</tr>
<tr>
<td>730000</td>
<td>790000</td>
</tr>
<tr>
<td>3100000</td>
<td>3300000</td>
</tr>
</tbody>
</table>

Generally, the larger the count the more accurate the sample, as would be expected.

3.6 Examples of Geospatial Query Plans

This section illustrates query optimization and query execution using the LOD2 geobench dataset. The geobench workload is expressed using four different, nearly equivalent formulations of the
queries and are explained in more detail in the LOD2 D5.1.2 deliverable.

This is adaptation of an Open Street Map extract of central Europe. The dataset in question is at scale 10, meaning 2.5 billion triples and 59 million geometries.

We consider the LOD2 geobench queries:

```sparql
select ?f as ?facet count(?s) as ?cnt
where {
  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f .
  filter(bif:st_intersects(bif:st_geomfromtext("BOX(0.3412 43.5141, 9.3412 48.0141)"), ?p))
} group by ?f order by desc(?cnt) limit 50
```

and:

```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 - 43.5141)/4.5) as ?x xsd:integer(40*(?o - 0.3412)/9) as ?y
where {
  {
    } union {
    } union {
    ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f . filter (?f = <http://linkedgeodata.org/ontology/Peak>)
    } union {
    ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f . filter (?f = <http://linkedgeodata.org/ontology/PostBox>)
    }
    ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?a ;
    <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o .
    filter(bif:st_intersects(bif:st_geomfromtext("BOX(0.3412 43.5141, 9.3412 48.0141)"), ?p))
    }
  }
} group by {?f ?x ?y
order by {?f ?x ?y
}
```

3 http://svn.aksw.org/lod2/D5.1.2/LOD2_D5_2.1_Initial_GEO_Benchmark_Definition.pdf
The first query counts the number of items of each type whose coordinates fall within a box.

The second query takes items that are of one of four selected types and are in a specified box and groups them by a rounded coordinate value. Each cell of the viewport has a count of items of each of the types.

In the first example, one will always begin with the geo condition, as there are no other conditions. One could also begin with the type predicate but there are substantially more triples with a rdf:type predicate than there are geometries in the experiment database.

In the second example, when at large zoom level, the condition on the type is nearly always more selective than the condition on the geometry. The query optimizer decides the join order and index utilisation based on sampling cardinalities of the predicates.

The case of the type being more selective than the geometry divides in two subclasses:

1. Fetch all the subjects of the type, fetch their geometries, fetch the geometry data from RDF_OBJ, check inclusion in the box.
2. Build a hash of the geometries in the box, scan the type and do an invisible hash join against the geometries.

Option 1 is better if there are very few of the type and very many geometries. Option 2 is better if there are many instances of the type and approximately as many geometries.

There exists a break-even point in the relative selectivities of each condition where one plan becomes better than the others. These break-even points have to be experimentally determined.

3.6.1 Execution Plan Profiles

We consider the execution profiles4 of two queries of Geobench method 3. Q1 selects the top 50 types of objects found in a large rectangle. Q2 selects, within the same rectangle, objects of diverse types and and divides the area into a grid. It then returns a count of items per type and grid cell.

Below are the SPARQL formulations and the Virtuoso cluster execution profiles. Comments are inline.

Query1:

```sparql
profile {'
sparql select ?f as ?facet count(?s) as ?cnt
where {
  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f .
  filter(bif:st_intersects(bif:st_geomfromtext("BOX(0.3412 43.5141, 9.3412 48.0141)"), ?p))
  group by ?f order by desc(?cnt) limit 50 ');
```

The query plan starts with the R tree index.

```text
{ 
  time 2.9e-05% fanout 1 input 1 rows
}
```

4 http://docs.openlinksw.com/virtuoso/databaseadmsrv.html#EfficientSQL
Subquery 27

```
{ time 1.9e-05% fanout 1 input 1 rows
  fork
  time 0.065% fanout 1 input 1 rows
  fork
  wait time 1.6e-07% of exec real time, fanout 0
}
```

The distributed execution starts. This is a DFG of two stages, the first is evaluated in parallel in all partitions, as the geo index is partitioned on the id of the geometry, not on its location.

```
QF {
  time 0.0071% fanout 0 input 0 rows
}
```

Stage 1

The geo index access is evaluated 48 times in parallel, once per each partition of the data. Its CPU cost is 19% of the total cycles used by the query. Each of the 48 invocation produces an average of 6.5e+04 results (fanout). These are the actual numbers. The cost model guess for the cardinality is 6.2e+06. The actual cardinality is 48 * 6.5e+04 = 3.12e06. The cost model guess was approximately 2 x greater than the actual.

```
time 19% fanout 6.5e+04 input 48 rows
geo 3 st_intersects (<c BOX2D(0,341200,43.514100,9.341200,48.014100,48.014100)>) node on DB.DBA.RDF_GEO 6.2e+06 rows
t4.O
```

For each geometry, the RDF subject is retrieved. This is 41% of CPU cycles.

```
time 41% fanout 1 input 3.1e+06 rows
RDF_QUAD_POGS 6.2e+06 rows(t4.S)
P = #geometry , O = cast
```

The execution switches to a different partition. So far the data was partitioned on the id of the geometry = the O of the triple with the geometry predicate. The next access gets the type of the subject, using an index that is partitioned on the S.

```
time 3.7% fanout 1 input 3.2e+06 rows
```
Stage 2

Given the S the type is fetched. We note that each S has an average of 3.1 types (fanout).

```
time   30% fanout   3.1 input  3.2e+06 rows
RDF_QUAD  6.8 rows(t2.O, t2.S)
inlined  P = #type , S = q_t4.S
```

After code:
```
0: isnotnull := Call isnotnull (t2.S)
5: BReturn 0
```

The group by and count is done on 99.9e06 rows of input.

The aggregation is done in each partition separately and the partitions are added up at the end. The type is a low cardinality property with a highly skewed distribution, hence a partitioning on the grouping key is not chosen by the SQL optimizer.

```
time   5.8% fanout       0 input  9.9e+06 rows
Sort (t2.O) => (isnotnull)
}
}
```

The rest of the plan reads the aggregation and sorts it by the count and returns the rows. This needs no further comment.

Below are statistics for the query execution as a whole. The CPU% is 3605%, of a theoretical maximum of 4800 (48 partitions of the data, 48 core threads). The CPU utilization is high because the first part of the query runs independently in each partition. The second stage of the plan is also well balanced, since the distribution of the subjects (3 million different ones) falls quite evenly on the partitions. The elapsed real time is 299 ms.

```
299 msec  3605% cpu,  6.3282e+06 rnd 1.0543e+07 seq 6.239e+06 same seg 85510 same pg
5113 messages  7704 bytes/m,  6.7% clw
Compilation: 1 msec 0 reads  0% read 0 messages  0% clw
```
Query 2:

```sparql
profile {'
  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 - 43.5141)/4.5) as ?y xsd:integer(40*(?o - 0.3412)/9) as ?x
      where 
      {
        
      } union {
      } union {
        ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f . filter (?f = <http://linkedgeodata.org/ontology/Peak>)
      } union {
        ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f . filter (?f = <http://linkedgeodata.org/ontology/PostBox>)
      } union {
        ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?a ;
        <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o .
        filter(bif:st_intersects(bif:st_geomfromtext("BOX(0.3412 43.5141, 9.3412 48.0141)")), ?p))
      })
    group by ?f ?x ?y
    order by ?f ?x ?y}
  }
};
```

The plan fetches all objects of each type,

```sparql
{ Subquery 41
```

Only one of the union terms is shown. The rest are similar.

```sparql
{ Union
}
wait time 3e-08% of exec real time, fanout 0
```
The query begins by considering all villages in the dataset. There are less villages than there are geometries in the rectangle, hence this makes sense.

We note that the first access to rdf quad is on the primary key, i.e. in order of S and partitioned by S. An index on O could have been used but this would have required a cross partition exchange (DFG stage). Hence the optimizer prefers the scan in order of S. The choice appears right due to the low CPU cost of the operation.

Stage 1

The geometry is retrieved. This is an RDF literal stored as a tagged geometry id in the O column. The actual geometry object is the RDF literals table.

Stage 2

The geo predicate selects approximately 1/30 of the input tuples.
The partition changes again, the next item to retrieve is the lat and long of the S. The CPU times are low since the operation is done on a small fraction of the original data.

```
time 0.69% fanout 0.99 input 1.3e+04 rows
Stage 3
time 0.51% fanout 1 input 1.3e+04 rows
RDF_QUAD 1.3 rows(t7.S, t7.O)
    inlined P = #lat , S = q_q_t6.S

time 0.42% fanout 1 input 1.3e+04 rows
RDF_QUAD 1.3 rows(t8.O)
    inlined P = #long , S = t7.S
```

At this stage, arithmetic follows for computing the squares in the grid. Deleted for brevity.

The query ends with an aggregation.
```
time 2.7% fanout 0 input 1.3e+04 rows
Sort (f, x, y) -> (inc, rdf_concat_impl)
```

At this point the plan has a similar fragment to the above for dealing with the other types of object requested. The 4 terms of the union are done in sequence. Each term individually is run with full parallelism. Since each term by itself can get full platform utilization there is no point running the terms in parallel since this would just lead to needless task switching. Each term of the union contributes to the same aggregation as the previous, since they all logically share the same group by. The group by's are added up at the end of the query. Again, the grouping keys are low cardinality hence partitioning the group by is not desired.

More of the same pattern repeats.
```
)
.
.
}
```

The rest of the plan reads the aggregation and returns the data. No comment needed.

We note a reasonably fast execution with reasonable platform utilization, about half the CPU% of the previous query. Less data was processed and there were more sequential steps, e.g. the 4 union terms, hence more time was spent on synchronizing barriers, leading to lower CPU%. Note the higher clw%.
This means cluster wait, meaning percentage of real time spent by the query coordinator process for peers to finish. Only time where the query coordinator has finished with its own work and is waiting for that of peers is counted. A higher value means less balanced load.

We further note that the query compilation took 20 ms, which is a noticeable fraction of the total time.

### 3.7 Scale Out and Data Location

A single server configuration has one R tree per geo index. A cluster configuration has one R tree for each slice of a geo index. The cluster case always partitions the R tree on the id of the geometry, i.e. the foreign key to the RDF_OBJ or other table containing the geometries. Therefore a query that begins with a geo lookup executes one thread per slice, all in parallel. A query that checks the geometry of a known object against a condition will send the operation to the partition of the object. Given a value of the O column of RDF_QUAD, one always knows which partition has the matching row of RDF_OBJ.

The partitioning of the R tree is the same as that of the RDF_OBJ table, by hash of the high bits of the RO_ID.

### 3.8 Execution Parallelism

Cluster plans that begin with a geo index lookup run one thread per slice, i.e. typically full platform since one normally makes a slice per core thread. Single server cases run one thread in the geo index and parallelize from the next operation on the pipeline onwards. For example, the R tree produces a million matches and the vector of a million matches is split into 16 smaller vectors, each served by its own thread. Before the splitting, the values are sorted so that each thread has a non-overlapping part of the index. This increases access locality and decreases contention. The contention in question consists of page lookup from buffer pool where two threads falling on the same page at the same time will have to enter the buffer one after the other. Many reading threads can be on the same buffer at a time, of course.

In the cluster case separating the trees for each slice into a completely independent data structure eliminates any possibility of inter-thread contention within a single query.

Unions (SQL union all) followed by aggregation are run with each subquery in the union sequentially. If the union produces data for aggregation, each subquery of the union is run multithreaded. The partial aggregations are merged at the end of the union. This is similar both single server and cluster.

### 3.9 System Prototype

The geospatial enhancements made as part of this task are available in the Virtuoso 7 open source [Git develop/7 branch](https://github.com/openlink/virtuoso-opensource/tree/develop/7). Instructions on compiling and building the Virtuoso open source code from Git is provided in the [Virtuoso open source Wiki](https://virtuoso.openlinksw.com/w/1.0).
A live Virtuoso instance where the Geospatial can be run from is available on the Virtuoso LOD Cloud server instance as this hosts the standard LinkedGeoData Datasets amongst most other in the LOD Cloud. Enabling "Query 1" described in the section above to be run against it is

```
select ?f as ?facet count(?s) as ?cnt
from <http://linkedgeodata.org>
where {
  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?f .
  filter(bif:st_intersects(bif:st_geomfromtext("BOX(0.3412 43.5141, 9.3412 48.0141)"), ?p))}
group by ?f order by desc(?cnt) limit 50 ')
```

returning the following results:

<table>
<thead>
<tr>
<th>facet</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://linkedgeodata.org/ontology/Amenity">http://linkedgeodata.org/ontology/Amenity</a></td>
<td>102107</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/HighwayThing">http://linkedgeodata.org/ontology/HighwayThing</a></td>
<td>95688</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/NaturalThing">http://linkedgeodata.org/ontology/NaturalThing</a></td>
<td>79297</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/Tree">http://linkedgeodata.org/ontology/Tree</a></td>
<td>73123</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/PowerThing">http://linkedgeodata.org/ontology/PowerThing</a></td>
<td>70484</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/PowerTower">http://linkedgeodata.org/ontology/PowerTower</a></td>
<td>63700</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/ManMadeThing">http://linkedgeodata.org/ontology/ManMadeThing</a></td>
<td>55337</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/SurveyPoint">http://linkedgeodata.org/ontology/SurveyPoint</a></td>
<td>51201</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/Place">http://linkedgeodata.org/ontology/Place</a></td>
<td>48982</td>
</tr>
<tr>
<td><a href="http://linkedgeodata.org/ontology/BusStop">http://linkedgeodata.org/ontology/BusStop</a></td>
<td>39986</td>
</tr>
</tbody>
</table>

Click on the "Query 1" link to see the live results.

---

6 http://www.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSGitUsage
7 http://lod.openlinksw.com/sparql
3.10 Conclusions and Future Work

We have implemented query optimizations that correspond to the present workload, e.g. parallel execution of union all, which is of course useful in a number of situations, including TPC-DS. We have identified further simple optimizations that will significantly accelerate the tile based access.

Platform utilization is generally good, with long running queries gaining from parallelism. Therefore the throughput increase with multiple concurrent sessions is less than linear, as a single session uses about between 1/2/3 and 1/2 of the platform. A 100% platform utilization for a single stream cannot be expected since queries remain short and consist of aggregation and re-aggregation which must eventually converge on a single thread. Also the cluster execution model has better parallelism for geo index operations than the single.

We will proceed with further benchmark comparison of RDF and SQL based GIS performance of Virtuoso compared with other geospatial data stores most notably PostGIS and possibly Oracle 12C Spatial. We expect an RDF penalty of 3-5x, commensurate with the observations on relational BI workloads.

---

9 http://www.tpc.org/tpcds/
4. References


USA, June 1984.


[OSM] OpenStreetMap project. Available at http://www.openstreetmap.org/


[PostGIS] PostGIS - Spatial and Geographic objects for PostgreSQL. Available at http://postgis.net/


[Virtuoso Column Store] Virtuoso, a Hybrid RDBMS/Graph Column Store - http://sites.computer.org/debull/A12mar/vicol.pdf