No Size Fits All – Running the Star Schema Benchmark with SPARQL and RDF Aggregate Views

Benedikt Kämpgen and Andreas Harth

Institute AIFB, Karlsruhe Institute of Technology, Karlsruhe, Germany
{benedikt.kaempgen,harth}@kit.edu

Abstract. Statistics published as Linked Data promise efficient extraction, transformation and loading (ETL) into a database for decision support. The predominant way to implement analytical query capabilities in industry are specialised engines that translate OLAP queries to SQL queries on a relational database using a star schema (ROLAP). A more direct approach than ROLAP is to load Statistical Linked Data into an RDF store and to answer OLAP queries using SPARQL. However, we assume that general-purpose triple stores – just as typical relational databases – are no perfect fit for analytical workloads and need to be complemented by OLAP-to-SPARQL engines. To give an empirical argument for the need of such an engine, we first compare the performance of our generated SPARQL and of ROLAP SQL queries. Second, we measure the performance gain of RDF aggregate views that, similar to aggregate tables in ROLAP, materialise parts of the data cube.

Keywords: Linked Data, OLAP, Star Schema Benchmark, View.

1 Introduction

Analytical queries using SPARQL on RDF have gained interest since large amounts of statistics have been published as Linked Data[1] and promise effective Extract-Transform-Load pipelines for integrating statistics. Online Analytical Processing (OLAP) has been proposed as a decision support method for analysing Linked Data describing *data cubes* [12,6].

OLAP engines translate OLAP queries into a target query language of a database storing the multidimensional data. The predominant way in industry is ROLAP since 1) it can be deployed on any of the widely-used relational databases, 2) industry-relevant data such as from accounting and customer relationship management often resemble star schemas [17] and 3) research has focused on optimising ROLAP approaches [15]. Instead of storing the data in a relational database, we have proposed to collect Statistical Linked Data reusing the RDF Data Cube Vocabulary (QB) and to transform OLAP into SPARQL queries [14]. Yet, there is little work on evaluating and optimising analytical

[1] `http://wiki.planet-data.eu/web/Datasets`

P. Cimiano et al. (Eds.): ESWC 2013, LNCS 7882, pp. 290–304, 2013.
© Springer-Verlag Berlin Heidelberg 2013

queries on RDF data [4,5]. We expect that, similar to general-purpose relational databases, a "one size fits all" [17] triple store will not scale for analytical queries. In this paper, we intend to give an empirical argument in favor of creating a specialised OLAP engine for analytical queries on Statistical Linked Data. Contributions of this paper are centered around four analytical query approaches listed in the following table:

	No Materialisation	Materialisation
Relational data / SQL	RDBMS / ROLAP	ROLAP-M
Graph data / SPARQL	OLAP4LD-SSB/-QB [14]	OLAP4LD-QB-M

- We compare the performance of traditional relational approaches (RDBMS / ROLAP) and of using a triple store and an RDF representation closely resembling the tabular structure (OLAP4LD-SSB). We compare those approaches with our OLAP-to-SPARQL approach [14] reusing a standard vocabulary for describing statistics (OLAP4LD-QB). To use a credible benchmark, we extend our approach for multi-level dimension hierarchies.
- We measure the performance gain of the common ROLAP query optimisation approach to precompute parts of the data cube and to store those "views" in aggregate tables, since they do not fit in memory [15,10] (ROLAP-M). We apply materialisation to our approach, represent views in RDF (OLAP4LD-QB-M) and evaluate their performance gain.

In Section 2, we introduce an OLAP scenario and present our OLAP-to-SPARQL approach. Our optimisation approach of using RDF aggregate views we present in Section 3. In Section 4, we evaluate both OLAP-to-SPARQL approach and RDF aggregate views. In Section 5, we discuss the results, after which we describe related work in Section 6 and conclude in Section 7.

2 OLAP-to-SPARQL Scenario and Approach

We now shortly introduce an OLAP scenario, taken from the Star Schema Benchmark (SSB) [16]. We then use this scenario to explain our extended OLAP-to-SPARQL approach [14] for multi-level hierarchies. In Section 4, we will use the scenario and benchmark for a performance evaluation.

SSB describes a data cube of lineorders. Any lineorder (fact) has a value (member) for six dimensions: the time of ordering (*dates*), the served *customer*, the product *part*, the *supplier*, the ordered *quantity* and granted *discount*. Depending on the member for each dimension, a lineorder exhibits a value for measures having a certain aggregation function with which to compute its value, e.g., *sum profit*, computed by *sum revenue* minus *sum supplycost*.

Dimensions exhibit hierarchies of levels that group members and relate them to higher-level members, e.g., dates can be grouped starting from the lowest *dateLevel* over *yearmonthLevel* to *yearLevel*. Since a week can be spread over two months or years, there is a separate hierarchy where dates can be grouped by *weeknuminyear*, e.g. "199322". Customers and suppliers can be grouped into

cities, nations, and regions and parts into brands, categories and manufacturers. Any hierarchy implicitly has a special-type *ALL* member, which groups all members into one special-type *ALL* level.

SSB provides a workload of 13 queries on the data cube. Each query is originally provided in SQL. For instance, *Q2.1* computes per year the revenues (in USD) for product brands from product category MFGR#12 and of suppliers from AMERICA. Results from this query usually are shown in pivot tables such as the following:

Year\Brand	MFGR#121	MFGR#1210	...	MFGR#129
1992	667,692,830	568,030,008	...	614,832,897
...	...	...	...	...
1998	381,464,693	335,711,347	...	319,373,807

Filter: partCategory = "categoryMFGR#12"
AND supplierRegion = "AMERICA"

More information about the benchmark we provide on our benchmark website [13]. In subsequent sections we present and compare different logical representations of the SSB data cube on Scale 1. First, we describe SSB using an extended OLAP-to-SPARQL approach and sets of multidimensional elements such as *Dimension* and *Cube* [14]. Then, we describe an engine that translates OLAP queries on SSB into SPARQL queries.

Member. All 3,094 dates, 30,280 customer, 201,030 part and 2,280 supplier members from each level are represented as URIs. Any member, e.g., *rdfh:categoryMFGR-35*, links to members on the next lower level via *skos:narrower*, e.g., *rdfh:brand1MFGR-3527*. 51 quantity and 11 discount members we encode as RDF Literal values. Also, we define URIs representing the special-type *ALL* member for each dimension, e.g., customer *rdfh:lo_custkeyAllAll*. Those *ALL* members will later be needed for representing aggregate views.

Level. Every level is represented as a URI, e.g., *rdfh:lo_orderdateDateLevel*, has a *xkos:depth* within its hierarchy and links to a set of members via *skos:member*. The vocabulary *XKOS*[2] allows to represent hierarchy levels.

Hierarchy. Each dimension has one or two (dates) hierarchies. Every hierarchy is represented as a URI, e.g., *rdfh:lo_orderdateCodeList*. Levels with a depth link to the hierarchy via *skos:inScheme*.

Dimension. Every dimension such as dates is represented as an object property, e.g., *rdfh:lo_orderdate* and defines its hierarchy via *qb:codeList*. The simple dimensions *quantity*, *discount* are represented as datatype properties.

Measures. Every measure such as the sum of revenues is represented as a datatype property, e.g., *rdfh:lo_revenue*. The component specification of a measure defines the aggregation function, e.g., SUM, via *qb4o:hasAggregateFunction*, as proposed by Etcheverry and Vaismann [7]. Since there is no recommended way to represent more complex functions, for formulas, we use String Literals using measure URIs as variables.

DataCubeSchema. The data cube schema of the SSB data cube is represented as an instance *rdfh-inst:dsd* of *qb:DataStructureDefinition* and defines the dimensions and measures of the data cube.

[2] `https://github.com/linked-statistics/xkos`

Fact. Every possible lineorder can be represented as a *qb:Observation*. Any observation links for each dimension property to the URI of a member or a Literal value (quantity, discount), and for each measure property to a Literal value. Whereas *base facts* with each dimension on the lowest level are given by the SSB dataset, aggregated facts on higher levels of dimensions of the cube need to be computed.

DataCube. The SSB data cube is identified by the dataset *rdfh-inst:ds*. The dataset defines the schema *rdfh-inst:dsd* and has attached via *qb:dataSet* all base facts.

All queries of SSB can be formalised as OLAP queries on single data cubes with multi-level hierarchies as per Definition 1, e.g., *Q2.1* as follows with abbreviated names: `({yearLevel, ALL, brand1Level, ALL, ALL, ALL}, {categoryLevel = categoryMFGR-12, s_regionLevel = s_regionAMERICA}, {lo_revenue})`. *Q2.1* slices dimensions customer, supplier, discount, quantity, rolls up dates to years and part to product brands, dices for a specific product part category and supplier region and projects the revenues.

Definition 1 (OLAP Query). *Given a data cube* $c = (cs, C) \in DataCube$*, with* $cs = (?x, D, M) \in DataCubeSchema$*,* $C \in 2^{Fact}$*.* $D = \{D_1, D_2, \ldots D_d\} \subseteq Dimension$ *is an ordered list of dimensions with a set of levels* $L_i = \{l_1, l_2, \ldots\} \subseteq Level$*, including the special-type* ALL *level. Each level* l_i *has* $memberno(l_i)$ *members.* $M \subseteq Measure$ *is an ordered list of measures. We define an OLAP query on this cube with* $OLAP\ Query = \mathcal{SC} \times 2^{Fact}$ *with* $(c, SlicesRollups, Dices, Projections) \in \mathcal{SC}$*, with* $SlicesRollups \subseteq L_1 \times L_2 \times \ldots \times L_d$ *a level for each dimension in the same order (for roll-ups), including the special-type level ALL (for slices), with Dices a set of conditional terms on members of levels (for dice) and with* $Projections \subseteq M$ *a set of selected measures from a data cube (for projection). An OLAP query results in a set of facts from the data cube.*

Given $Member$, $Level$, $Hierarchy$, $Dimension$, $Measure$, $DataCubeSchema$, $Fact$, and $DataCube$ as sets of multidimensional data, we define OLAP Engine $\subseteq$ OLAP Query $\times$ Target Query with OLAP Query as per Definition 1, Target Query a query in a target query language such as SQL and SPARQL. The following pseudocode algorithm implements an OLAP engine that transforms an OLAP query into a SPARQL query. The algorithm separately creates the WHERE, SELECT and GROUP BY clause. Note, in this pseudocode we disregard translating multidimensional elements to URI representations and variables, more efficient filters, complex measures and ordering:

```
1  Algorithm 1: OLAP-to-SPARQL
2  Input: OLAP Query (cube, SlicesRollups, Dices, Projections)
3  Output: SPARQL query string
4  begin
5    whereClause = "?obs qb:dataSet " + cube.ds.uri.
6    for level ∈ SlicesRollups do
7      levelHeight = level.getHeight()
8      dimension = level.getHierarchy().getDimension()
9      dimVar = makeUriToParameter(dimension)
10     hashMap.put(dimension, levelHeight)
11     for i = 0 to levelHeight - 1 do
```

```
12        rollUpsPath += dimVar + i + ". " + dimVar + i + " skos:narrower "
13      whereClause += "?obs " + dimension.uri + rollUpsPath + dimVar +
            levelHeight + ". "
14      whereClause += dimVar + levelHeight + " skos:member " + level.uri
15      selectClause, groupByClause += " "+ dimVar + levelHeight
16    for member ∈ Dices.getPositions().get(0).getMembers() do
17      if (diceslevelHeight > slicesRollupsLevelHeight) do
18        dicesLevelHeight = member.getLevel().getHeight()
19        slicesRollupsLevelHeight = hashMap.get(dimension)
20        dimension = member.getLevel().getHierarchy().getDimension()
21        dimVar = makeUriToParameter(dimension)
22        for i = slicesRollupsLevelHeight to dicesLevelHeight - 1 do
23          dicesPath += dimVar + i + ". " + dimVar + i + " skos:narrower "
24        whereClause += "?obs " + dimension.uri + dicesPath + dimVar +
              dicesLevelHeight + ". "
25    whereClause += " Filter("
26    for position ∈ Dices.getPositions() do
27      for member ∈ position.getMembers() do
28    dimVar =
         makeUriToParameter(member.getLevel().getHierarchy().getDimension())
29    memberFilterAnd += "AND " + dimVar + diceslevelHeight + " = " + member
30      memberFilterOr += "OR " + memberFilterAnd
31    whereClause += memberFilterOr + ") "
32    for measure ∈ Projections do
33      measVar = makeUriToParameter(measure)
34      selectClause += measure.getAggregationFunction() + "(" + measVar +
            ") "
35      whereClause += " ?obs " + measure.uri + " " + measVar + " ."
36    return selectClause + whereClause + groupByClause
```

We query for all observations of the cube (line 5). Then, for each level, we create a property path starting with ?obs and ending with a dimension variable at the respective level (line 6 to 14). Each level height we store in a map in order to later check whether graph patterns need to be added for dices (10). Then, we add the variables to the select and group by clause (15). Now, we add graph patterns for dices (16 to 31). We assume that the set of conditional terms on members of levels, *Dices*, can be translated into a set of positions with each position describing a possible combination of members for each diced dimension (16). Diced dimensions and levels are fixed for each position; therefore, we only use the first position for adding graph patterns (16). We assume furthermore that measures are only contained in *Projections* but not *SlicesRollups* and *Dices*. We only need to add graph patterns if the height of the diced level is larger than the level mentioned for the same dimension in *SlicesRollups* (17). Then, from the positions in *Dices*, we filter for one (OR, 30) of all possible combinations (AND, 29) of members for each diced dimension. Finally, for each measure in *Projections*, we add a variable with the aggregation function of the measure to the select clause and graph patterns to the where clause (34,35). The following listing shows the relevant parts of the SPARQL query for *Q2.1*:

```
1  SELECT ?rdfh_lo_orderdate ?rdfh_lo_partkey1 sum(?rdfh_lo_revenue) as
       ?lo_revenue
2  WHERE {
3  ?obs qb:dataSet rdfh-inst:ds; rdfh:lo_orderdate ?rdfh_lo_orderdate0.
4  ?rdfh_lo_orderdate1 skos:narrower ?rdfh_lo_orderdate0.
5  ?rdfh_lo_orderdate2 skos:narrower ?rdfh_lo_orderdate1.
6  ?rdfh_lo_orderdate skos:narrower ?rdfh_lo_orderdate2.
7  rdfh:lo_orderdateYearLevel skos:member ?rdfh_lo_orderdate.
8  ?obs rdfh:lo_partkey ?rdfh_lo_partkey0.
9  ?rdfh_lo_partkey1 skos:narrower ?rdfh_lo_partkey0.
10 ?rdfh_lo_partkey skos:narrower ?rdfh_lo_partkey1.
```

```
11  rdfh:lo_partkeyCategoryLevel skos:member ?rdfh_lo_partkey.
12  ?obs rdfh:lo_suppkey ?rdfh_lo_suppkey0.
13  ?rdfh_lo_suppkey1 skos:narrower ?rdfh_lo_suppkey0.
14  ?rdfh_lo_suppkey2 skos:narrower ?rdfh_lo_suppkey1.
15  ?rdfh_lo_suppkey skos:narrower ?rdfh_lo_suppkey2.
16  rdfh:lo_suppkeyRegionLevel skos:member ?rdfh_lo_suppkey.
17  ?obs rdfh:lo_revenue ?rdfh_lo_revenue.
18  FILTER(?rdfh_lo_partkey = rdfh:lo_partkeyCategoryMFGR-12 AND
        ?rdfh_lo_suppkey = rdfh:lo_suppkeyRegionAMERICA ).
19  } GROUP BY ?rdfh_lo_orderdate ?rdfh_lo_partkey1 ORDER BY
        ?rdfh_lo_orderdate ?rdfh_lo_partkey1
```

Here, *Dices*, {`categoryLevel = categoryMFGR-12, s_regionLevel = s_regionAMERICA`}, is translated into one position with one member for part category level and one member for supplier region level. The SPARQL query queries for all facts within the data cube (line 3), adds *skos:narrower* paths up to yearLevel, categoryLevel and s_regionLevel (4 to 16), selects lo_revenue as measure (17), filters for a certain member of part category and of supplier region (18) and groups by yearLevel and brand1Level (19). We assume all RDF data stored in a default graph.

3 RDF Aggregate Views

We now apply a common optimisation technique to the OLAP engine implementing our OLAP-to-SPARQL approach: data cube materialisation, i.e., precomputing of certain facts from the entire data cube and storing them for reuse.

Just as Harinarayan et al. [11], we assume that the cost of answering an OLAP query is proportional to the number of facts that need to be scanned, e.g., for validating a filter or calculating an aggregation. So far, any OLAP query to the SSB data cube needs to scan the 6,000,000 base facts. Intuitively, materialisation pre-computes facts with dimensions on higher levels, so that views contain fewer and already aggregated facts to be examined for filtering or further aggregation.

Definition 2 (Aggregate View). *We define a view in a data cube c as an OLAP query per Definition 1* $(c, SlicesRollups, Dices, Projections)$ *with* $SlicesRollups \subseteq L_1 \times L_2 \times \ldots \times L_d$*, Dices the empty set, and Projections a set of measures. Thus, any fact within the view gives a value for each of its measures for a certain combination of level members. A view may be sparse and not contain facts for each possible combination of members. The maximum number of facts within a view is given by* $\prod memberno(l_i), l_i \in L_i$*. The number of views in the data cube is given by* $\prod |L_i|$*. The facts from an aggregate view can be generated by executing the OLAP query using an OLAP engine.*

The SSB data cube contains $6*5*5*5*2*2 = 3,000$ views with dates having six levels since the two hierarchies of dates contain the same lowest and *ALL* level. The advantage of aggregate views as per Definition 2 is that the entire set of views of a data cube with multi-level hierarchies can be represented as a *data cube lattice* [11], see Figure 1 for an illustration of the lattice of the SSB cube. Any view is represented by the level of each dimension, omitting any *ALL* levels. The single view on the lowest level corresponds to the OLAP query that contains all

base facts, i.e., the view returns all non-aggregated facts from the SSB dataset. The view contains maximum $2,555 * 30,000 * 200,000 * 2,000 * 51 * 11 \geq 1.7 * 10^{19}$ facts, however, SSB provides a sparse data cube with $6,000,000$ facts. From this lowest view one can reach higher views via *roll-up* operations on dimensions, e.g., the next higher view on the right side rolls up to the *ALL* level of quantity. The single view on the highest level in Figure 1 corresponds to the OLAP query that returns one single fact grouping by the special-type level *ALL* with the single member *ALL* for each dimension.

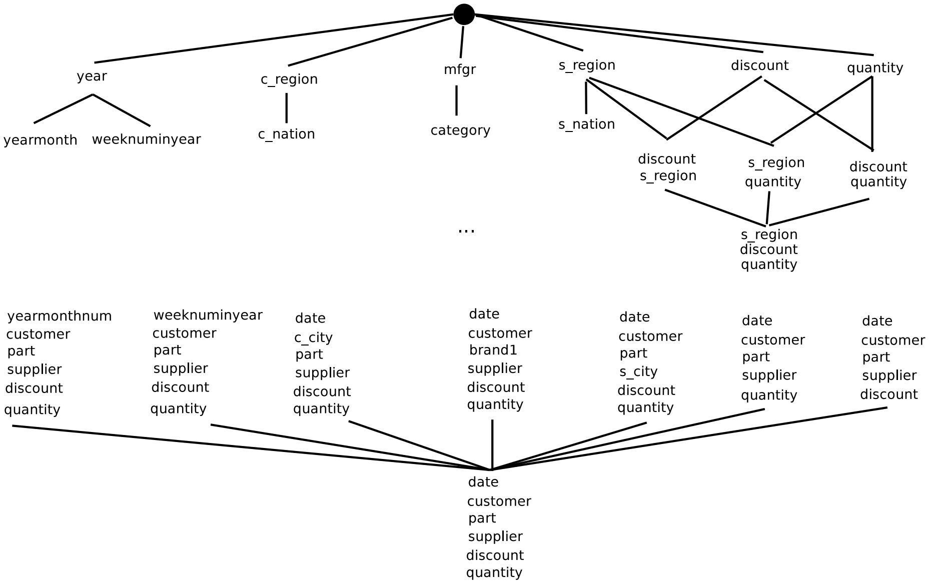

Fig. 1. Illustration of data cube lattice of SSB

The higher the view on a path in the lattice, the fewer facts it contains, since higher levels group lower-level members into groups of fewer members. For distributive aggregation functions such as *SUM*, and algebraic formulas such as `SUM(rdfh:lo_revenue - rdfh:lo_supplycost)`, we do not run into summarisability problems [9] and a view can be computed from any view on a lower level that can be reached via a *roll-up* path; for instance, the view grouping by quantity on the right upper corner can be computed from the view grouping discount and quantity, s_region and quantity, their collective child view grouping by s_region, discount, and quantity as well as from any other reachable lower level view not displayed. The holistic aggregation function `SUM(rdfh:lo_extendedprice * rdfh:lo_discount)` is not further aggregated from views, thus, *Q1.1* to *Q1.3* return correct results.

Summing up for each dimension the number of members on the lowest level, the numbers of members on each level per hierarchy, and the special-type member *ALL*, we can calculate the maximum number of facts in the entire data

cube: $3095 * 30281 * 201031 * 2281 * 52 * 12 > 2.6 * 10^{19}$. As materialising the entire data cube would 1) take too much time and 2) require too much hard disk space, we are concerned with deciding which views to materialise. We define for a given OLAP query (c, *SlicesRollups*, *Dices*, *Projections*) as per Definition 1 a single *closest* view in the lattice from which we can create the results by only scanning the facts in the view [11]: We create a view (c, *SlicesRollups'*, *Dices'*, *Projections'*) on the same cube that contains in *SlicesRollups'* for each dimension the lowest level mentioned in *SlicesRollups* and *Dices*, contains an empty set for *Dices'* and $M' = M$. The following term describes the *closest* view for *Q2.1*, the other views are translated, accordingly: `({yearLevel, ALL, brand1Level, s_regionLevel, ALL, ALL}, ∅, {lo_revenue})`. The view contains maximum 35,000 facts and as such is considerably smaller than the SSB dataset with 6,000,000 facts. Note, *Q2.2* and *Q2.3* can use the same view as *Q2.1* and *Q3.3* can use the same view as *Q3.2*, resulting in less time and less space for creating the views. Though some views can contain as many facts as there are base facts in the data cube, they often do not due to sparsity, e.g., *Q4.3* with 4,178,699 facts. For views may still be large, in ROLAP, views are stored in *aggregate tables*. Similarly, we represent views as *RDF aggregate views* reusing QB and store the triples together with the other multidimensional data in the same triple store. See Figure 2 for an illustration of this approach for *Q2.1*.

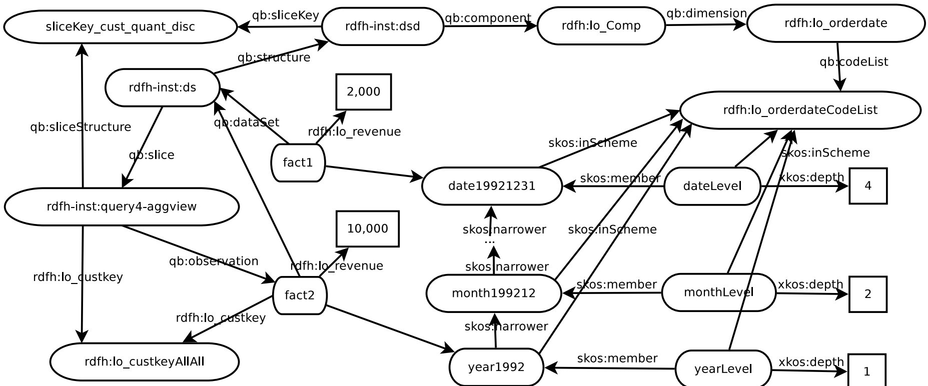

Fig. 2. Modelling RDF aggregate view rolling up to year level using QB

For dimensions on the *ALL* level, aggregate views only contain facts that fix those dimensions to the *ALL* member, e.g., *Q2.1* fixes customer. Therefore, we can represent *RDF aggregate views* as instances of *qb:Slice*, e.g., *rdfh-inst:query4-aggview*. *qb:SliceKey*s describe the structure of a slice, i.e., the sliced dimensions (not shown in figure). Slices explicitly state to what member a sliced dimension is fixed, e.g., *rdfh:lo_custkey* to *ALL*. In addition to base facts, e.g., *fact1* with member *date19921231* in the date level, facts are created that aggregate on the specific levels of the view. For instance, the view of Q2.1 contains via

qb:observation a *fact2* that rolls-up to the higher-level-member *year1992* in the year level of the dates hierarchy. The higher-level-member is connected to the lower-level-member in a *skos:narrower* path. Also, *fact2* rolls-up to the special-type *ALL* member of customer. The datatype property *xkos:depth* states for each level the depth of a level starting with 0 from the (implicit) *ALL* level. The following listing shows the relevant parts of a SPARQL INSERT query on the SSB data that populates the RDF aggregate view for *Q2.1*:

```
1   INSERT {
2   rdfh-inst:query4-aggview qb:observation _:obs.
3   _:obs rdfh:lo_orderdate ?d_year; rdfh:lo_custkey rdfh:lo_custkeyAllAll;
        rdfh:lo_partkey ?p_brand1; rdfh:lo_suppkey ?s_region;
        rdfh:lo_quantity rdfh:lo_quantityAllAll; rdfh:lo_discount
        rdfh:lo_discountAllAll; rdfh:lo_revenue ?lo_revenue.}
4   WHERE {{
5   SELECT ?d_year ?p_brand1 ?s_region sum(?rdfh_lo_revenue) as ?lo_revenue
        WHERE {
6   ?obs qb:dataSet rdfh-inst:ds.
7   ?obs rdfh:lo_orderdate ?d_date.
8   ?d_yearmonthnum skos:narrower ?d_date.
9   ?d_yearmonth skos:narrower ?d_yearmonthnum.
10  ?d_year skos:narrower ?d_yearmonth.
11  rdfh:lo_orderdateYearLevel skos:member ?d_year.
12  ?obs rdfh:lo_partkey ?p_part.
13  ?p_brand1 skos:narrower ?p_part.
14  rdfh:lo_partkeyBrand1Level skos:member ?p_brand1.
15  ?obs rdfh:lo_suppkey ?s_supplier.
16  ?s_city skos:narrower ?s_supplier.
17  ?s_nation skos:narrower ?s_city.
18  ?s_region skos:narrower ?s_nation.
19  rdfh:lo_suppkeyRegionLevel skos:member ?s_region.
20  ?obs rdfh:lo_revenue ?rdfh_lo_revenue.
21  } GROUP BY ?d_year ?p_brand1 ?s_region
22  }}
```

Here, we first create a SELECT query using our OLAP-to-SPARQL algorithm on the OLAP query (line 5), then this SELECT query is made a subquery of an INSERT query. Observations roll-up to members of specific levels and fix sliced dimensions (3). Resulting triples are stored in the default graph. We can now easily adapt our OLAP-to-SPARQL algorithm to use for an OLAP query the RDF aggregate views instead of the base facts from the SSB dataset. The following listing shows the SPARQL query for *Q2.1*.

```
1   SELECT ?d_year ?p_brand1 sum(?rdfh_lo_revenue) as ?lo_revenue
2   WHERE {
3   rdfh-inst:ds qb:slice ?slice.
4   ?slice qb:observation ?obs;
5       rdfh:lo_custkey rdfh:lo_custkeyAllAll;
6       rdfh:lo_quantity rdfh:lo_quantityAllAll;
7       rdfh:lo_discount rdfh:lo_discountAllAll.
8   ?obs rdfh:lo_orderdate ?d_year.
9   rdfh:lo_orderdateYearLevel skos:member ?d_year.
10  ?obs rdfh:lo_partkey ?p_brand1.
11  ?p_category skos:narrower ?p_brand1.
12  rdfh:lo_partkeyCategoryLevel skos:member ?p_category.
13  ?obs rdfh:lo_suppkey ?s_region.
14  rdfh:lo_suppkeyRegionLevel skos:member ?s_region.
15  ?obs rdfh:lo_revenue ?rdfh_lo_revenue.
16  FILTER(?p_category = rdfh:lo_partkeyCategoryMFGR-12 AND ?s_region =
        rdfh:lo_suppkeyRegionAMERICA ).
17  } GROUP BY ?d_year ?p_brand1 ORDER BY ?d_year ?p_brand1
```

Here, we query for observations from slices of *rdfh-inst:ds* that fix customer, quantity and discount to the *ALL* member, as indicated in *SlicesRollups* of *Q2.1* (lines 3 to 7). In comparison to the OLAP SPARQL query of *Q2.1* without views, we have a reduced set of triple patterns for rolled-up dimensions (*skos:narrower* paths) (8 to 14). To distinguish between observations from views slicing the same dimensions but rolling-up to different levels, we require for each member the correct level (9, 12, 14). And we add filters on diced dimensions (16).

4 Evaluation

We now give an overview of tested approaches and the reasons for their selection, then explain the design of the tests. See the benchmark website for this paper [13] for more background information about the tests:

Name	Data Format	Metadata	Query Language	Engine/Database	Pre-processing (s)	Rows / Triples
RDBMS	Relational	-	SQL	MySQL	22	6,234,555
ROLAP-M	Relational	XML	SQL	MySQL, Mondrian	4,507	14,975,472
OLAP4LD-SSB	Graph-based	-	SPARQL	Open Virtuoso	5,352	108,021,078
OLAP4LD-QB	Graph-based	RDF/QB	SPARQL	Open Virtuoso	5,744	116,832,479
OLAP4LD-QB-M	Graph-based	RDF/QB	SPARQL	Open Virtuoso	26,032	190,060,632

RDBMS and ROLAP-M represent the traditional approaches with a widely-used Open-Source relational database (MySQL 5.1 v5.1.63) and SQL. ROLAP-M uses aggregate tables for optimising queries. The other tests represent graph-based approaches with a widely-used Open-Source triple store (Open Virtuoso v06.01.3127) and SPARQL 1.1 for aggregate and sub-queries. Whereas OLAP4-LD-SSB represents SSB data without reusing a vocabulary, OLAP4LD-QB reuses QB which allows us to materialise parts of the data cube as RDF aggregate views in OLAP4LD-QB-M.

We use the Star Schema Benchmark [16], since SSB 1) refines the decision support benchmark TPC-H by deriving a pure star schema from the schema layout in order to evaluate analytical query engines [1], and 2) can be regarded as a realistic data source since statistics published as Linked Data are typically highly structured [4,3]. We run each approach on a Debian Linux 6.0.6, 2x Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz with 16 cores, 128GB RAM and 900GB RAID10 on 15k SAS disks for local data storage. We assume unlimited amount of space but configure the databases to only use an amount of memory clearly below 100% of the space the data files surmount to (400M for relational approaches, < 650M for graph-based approaches), since storing all multidimensional data in main memory is often too costly. For each approach we 1) translate the SSB data cube at Scale 1 with 6,000,000 lineorders into the respective data format for storage in the database, 2) simulate an OLAP engine translating the SSB OLAP queries into the respective query language of the database 3) before each test, shut-down all other applications not needed and run the test once to populate the disk cache (warm-up) and 4) document the elapsed query time of each query in turn. Note, we do not consider data refreshes. For running the SSB benchmark and collecting the data about elapsed query times, we used the Business Intelligence Benchmark (BIBM)[3]. BIBM also ensured identical results

[3] `http://sourceforge.net/projects/bibm/`

for the approaches through qualification files (provided at website). We now describe for each approach how we stored SSB data in the database and translated SSB OLAP queries to the database query language.

RDBMS. We created a schema file for dimension and fact tables and populated the database with an SSB data generator. We setup column data types as recommended by SSB and indexes for each foreign key in the fact table, primary keys for the fact table comprising the dimension keys and primary keys for dimension tables in a standard Star Schema fashion. Loading of 6,234,555 rows of data took 22s. The SQL queries of SSB could be reused with minor MySQL-syntax-specific modifications. We switched off query cache so that MySQL after a warm-up would not read all queries from cache. Note, we have compared those SQL queries with SQL queries created by the widely-used Open-Source ROLAP engine Mondrian (v3.4.1). Mondrian stores data cube metadata in XML and would for example deliberately query for more data than requested by the query and cache the results for later use; however, SSB minimises overlap between queries, e.g., *Q1.1* uses discounts between 1 and 3, *Q2.1* between 4 and 6. Since the performance gain of using Mondrian-created SQL queries instead of the original SSB SQL queries showed small, we only include a Mondrian test in the benchmark website (ROLAP).

ROLAP-M. We created aggregate tables without indices and keys for the *closest* view to each query using SQL INSERT queries on the original tables from *RDBMS*. Time included 22s for preparing *RDBMS* with 6,234,555 rows and 4,485s for creating the aggregate tables with another 8,740,917 rows. For each OLAP query we created an SQL query using the *closest* aggregate table. Similarly, Mondrian would choose the aggregate table with the smallest number of rows and create an SQL query with comparable performance.

OLAP4LD-SSB. With BIBM we translated the SSB tabular data into RDF/TTL files using a vocabulary that strongly resembles the SSB tabular structure: A lineorder row is represented as a URI which links for each dimension via an object property, e.g., *rdfh:lo_orderdate*, to a URI representing a row from the respective dimension table, e.g., *rdfh:lo_orderdate19931201*. From this URI, datatype properties link to Literal values for members, e.g., month “199312”. Quantity and discount are directly given using datatype properties from a lineorder. Each measure is attached to the lineorder URI using a datatype property. Translation took 48s, bulk loading of $108,021,078$ triples 5,304s. For each SSB OLAP query, we tried to build the most efficient SPARQL-pendant to the original SSB SQL queries, e.g., reducing the number of joins.

OLAP4LD-QB. We created RDF metadata for the SSB data cube using our extended OLAP-to-SPARQL approach and via a small script added links from each lineorder of *OLAP4LD-SSB* to *rdfh-inst:ds*. Using SPARQL INSERT queries for each dimension, we grouped dimension members into levels of hierarchies, and added them to the triple store. Creating the *OLAP4LD-SSB* data and adding links took 48s and 38s, the INSERT queries 14s; compressing and bulk loading

of $116,832,479$ triples took 60s and 5,584s. Simulating our OLAP-to-SPARQL algorithm, we manually translated the SSB queries to SPARQL.

OLAP4LD-QB-M. For each SSB query, we created a *closest* RDF aggregate view using a SPARQL INSERT query. Setting up *OLAP4LD-QB* took 5,744s, the SPARQL INSERT queries 20,288s for another $73,228,153$ triples. Also, we created SPARQL queries that use the *closest* views.

5 Presentation and Discussion of Results

In this section, we evaluate 1) the scalability of the OLAP-to-SPARQL approach and 2) the performance gain of RDF aggregate views. Table 1 lists performance-relevant SSB query features. *Filter factor* measures the ratio of fact instances that are filtered and aggregated. Filter factors are computed by multiplying the filter factors of each dice, e.g., for *Q2.1* the filter factor is 1/25 for part times 1/5 for supplier. *View factor* measures the ratio of fact instances that are contained in a view in relation to the 6M base facts. For example, from the filter factor and view factor, we see that query flight 4 (Q4) iteratively drills-down to more granular levels (up to 4,178,699 facts) but filters for fewer, more specific lineorders. With *RDBMS joins* we describe the number of joins between tables in the SQL representation of a query. Note, ROLAP-M does not need joins. With *SSB, QB and QB-M joins* we state the number of triple pattern joins, pairs of triple patterns mentioning the same variable. Table 2 lists the elapsed query times (s) which we now discuss.

Table 1. Overview of SSB queries and their performance-relevant features

Feature	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Filter factor	.019	.00065	.000075	.008	.0016	.0002	.034	.0014	.000055	.00000076	.016	.0046	.000091
View factor	.00064	.0073	.0032	.0058	.0058	.0058	.0007	.0728	.0728	.5522	.0007	.0036	.6964
RDBMS joins	1	1	1	3	3	3	3	3	3	3	4	4	4
SSB joins	5	5	6	8	7	7	9	9	7	8	10	12	12
QB joins	8	6	6	15	13	14	16	16	16	12	22	22	22
QB-M joins	9	9	9	12	11	11	13	13	11	12	13	14	14

Table 2. Evaluation results with single and total elapsed query time (s)

Name	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3	Total
RDBMS	1.6	1.1	1.1	16.1	15.7	15.4	10.4	7.8	7.6	3.1	11.0	5.3	5.0	101
ROLAP-M	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.6	0.1	0.1	0.8	2
OLAP4LD-SSB	22.5	0.8	0.2	16.1	0.9	0.2	28.5	2.1	1.0	0.4	N/A	36.8	9.6	119
OLAP4LD-QB	46.1	1.3	0.2	55.0	49.4	31.1	145.7	12.5	1.8	87.2	175.3	544.5	24.9	1175
OLAP4LD-QB-M	19.9	10.2	10.2	366.3	356.7	356.3	468.5	467.6	4.6	4.6	0.1	0.4	55.4	2121

ROLAP-M overall is 50 times faster than *RDBMS* for not requiring any joins and a reduced number of facts to scan for aggregation. Whereas RDBMS has to first scan 6M rows and then to aggregate, ROLAP-M only has to scan the view and to aggregate from there. Affirmatively, the views of Q3.4 and Q4.3 with very low selectivity show smaller benefits. However, preparing ROLAP-M takes 200 times longer than RDBMS.

Comparing *OLAP4LD-SSB* and RDBMS, we see that the graph database is as fast as the relational alternative for some of the queries (e.g., Q1.2, Q2.1), slower for other queries (e.g., Q1.1, Q3.1, Q4.2), and even faster for others (Q2.2, Q3.3). Over all queries, OLAP4LD-SSB is only slightly worse, however, Q4.1 for no known reason does not successfully complete. Differences can be explained by the number of joins; for instance, whereas RDBMS requires for Q3.1 and Q4.2 three and four joins, OLAP4LD-SSB requires nine and twelve joins, respectively. If the number of joins is less divergent, differences can be explained by the filter factor and the fact that after filtering facts still need to be aggregated. In general, the smaller the filter factor, the better the graph database seems in comparison to the relational database, for instance Q2.2, Q2.3 and Q3.3. For low-selective queries, the graph database performs worse, e.g., Q1.1, Q3.1, Q4.2. This aligns with our expectations that a graph database is more optimised for high-selectivity metadata queries without aggregations. OLAP4LD-SSB requires 243 times as much time for loading.

OLAP4LD-QB reusing QB requires up to twice as many joins than OLAP4LD-SSB (Q2.3), since hierarchies are explicitly represented through *skos:narrower* paths from higher-level to lower-level members, and consequently is 10 times slower. Both approaches require similar pre-processing time. Yet, only OLAP4LD-QB can represent hierarchies and be optimised using RDF aggregate views.

Although *OLAP4LD-QB-M* overall leads to 1.8 times slower queries and performs considerably worse for query flights 2 and 3 (Q2/3), it succeeds in optimising query flight 4 (Q4). Similar to ROLAP-M, the performance gain RDF aggregate views can be explained by a reduced number of joins, e.g., for Q4.1, Q4.2. However, for most queries OLAP4LD-QB-M performs worse, since RDF aggregate views – different from ROLAP-M with separately created aggregate tables – are stored in the same graph and do not reduce the number of facts scanned for a query. Thus, whereas OLAP4LD-QB needs to scan for 6M facts, OLAP4LD-QB-M also needs to scan over facts from the aggregate views, in total 14.98M facts. Queries need to compensate for the increased effort in scanning by the reduced number of joins, in which Q2.1 to Q3.2 apparently do not succeed.

6 Related Work

In this section, we describe related work on 1) evaluating analytical query execution on RDF data, 2) representing multidimensional data as RDF and 3) materialising aggregate views on RDF data.

In our OLAP-to-SPARQL approach we have chosen RDF reusing QB as a logical representation, SPARQL as a query language for computation, and materialised *closest* views from the data cube lattice that promise the largest performance gain. We compare analytical queries on RDF with common alternatives in a realistic scenario, according to Erling [4] a prerequisite for successful RDF use cases and targeted optimisations [5]. Most notably, the *Berlin SPARQL*

Benchmark BI Use Case allows quantifying analytical query capabilities of RDF stores, but, so far, no work compares the RDF performance with the industry-standard of relational star schemas.

Recent work discusses approaches to represent multidimensional data in RDF [12,7,6], however, no approach deals with the computation and selection of data cube slices and dices in RDF, in particular, considering the special-type *ALL* members and levels for uniquely identifying all possible facts of a data cube.

Several authors discuss views over RDF data. Castillo and Leser [2] have presented work on automatically creating views from a workload of SPARQL queries. In the evaluation, they use a dataset with 10M triples and disregard queries that exhibit a high selectivity. Also, Goasdoué et al. [8] have discussed the creation and selection of RDF views. Their evaluation is done on a 35M triple dataset. In contrast to these approaches, our approach considers more complex views based on aggregation functions and hierarchies, materialises views as RDF reusing QB in a triple store and evaluates the applicability for high- and low-selectivity queries on a $>$ 100M triple dataset.

7 Conclusion

We now give an empirical argument in favor of creating a specialised OLAP engine for analytical queries on RDF. Although a triple store has shown almost as fast as a relational database, OLAP scenarios such as from the Star Schema Benchmark used in our evaluation require results in seconds rather than minutes. Materialised views with aggregate tables overall reach 50 times faster queries. Queries by our OLAP-to-SPARQL approach on data reusing the RDF Data Cube Vocabulary (QB) overall are 10 times slower than queries on data without reusing QB, for a large number of joins are required for rolling-up on dimensions; yet, only QB metadata allows to explicitly represent dimension hierarchies and to materialise parts of the data cube. RDF aggregate views show the capability to optimise query execution, yet, overall still take six times longer for preprocessing and not nearly reach the performance gain of aggregate tables in ROLAP. The reason seems that the reduced number of joins for queries on RDF aggregate views often cannot compensate for the increased number of facts that are stored in the triple store and need to be scanned for query execution. We conclude that the query optimisation problem intensifies in many OLAP scenarios on Statistical Linked Data and that OLAP-to-SPARQL engines for selection and management of RDF aggregate views are needed.

Acknowledgements. This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under the SFB/TRR 125 - Cognition-Guided Surgery and under the Software-Campus project. We thank Günter Ladwig, Andreas Wagner and the anonymous reviewers for helpful support and feedback.

References

1. Bog, A., Plattner, H., Zeier, A.: A mixed transaction processing and operational reporting benchmark. Information Systems Frontiers 13, 321–335 (2011)
2. Castillo, R., Leser, U.: Selecting Materialized Views for RDF Data. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010 Workshops. LNCS, vol. 6385, pp. 126–137. Springer, Heidelberg (2010)
3. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and Oranges: a Comparison of RDF Benchmarks and Real RDF Datasets. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (2011)
4. Erling, O.: Directions and Challenges for Semdata. In: Proceedings of Workshop on Semantic Data Management (SemData@VLDB 2010) (2010)
5. Erling, O.: Virtuoso, a Hybrid RDBMS/Graph Column Store. IEEE Data Eng. Bull. 35, 3–8 (2012)
6. Etcheverry, L., Vaisman, A.A.: Enhancing OLAP Analysis with Web Cubes. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) ESWC 2012. LNCS, vol. 7295, pp. 469–483. Springer, Heidelberg (2012)
7. Etcheverry, L., Vaisman, A.A.: QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In: Proceedings of the Third International Workshop on Consuming Linked Data (2012)
8. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View Selection in Semantic Web Databases. PVLDB 5, 97–108 (2011)
9. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Data Mining and Knowledge Discovery 1, 29–53 (1997)
10. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. In: Materialized Views. MIT Press (1999)
11. Harinarayan, V., Rajaraman, A.: Implementing Data Cubes Efficiently. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (1996)
12. Kämpgen, B., Harth, A.: Transforming Statistical Linked Data for Use in OLAP Systems. In: Proceedings of the 7th International Conference on Semantic Systems (2011)
13. Kämpgen, B., Harth, A.: Benchmark Document for No Size Fits All – Running the Star Schema Benchmark with SPARQL and RDF Aggregate Views (2012), `http://people.aifb.kit.edu/bka/ssb-benchmark/`
14. Kämpgen, B., O'Riain, S., Harth, A.: Interacting with Statistical Linked Data via OLAP Operations. In: Proceedings of Workshop on Interacting with Linked Data (2012)
15. Morfonios, K., Konakas, S., Ioannidis, Y., Kotsis, N.: ROLAP Implementations of the Data Cube. ACM Computing Surveys 39 (2007)
16. O'Neil, P., O'Neil, E., Chen, X.: Star Schema Benchmark - Revision 3. Tech. rep., UMass/Boston (2009), `http://www.cs.umb.edu/~poneil/StarSchemaB.pdf`
17. Stonebraker, M., Bear, C., Cetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., Zdonik, S.: One Size Fits All? – Part 2: Benchmarking Results. In: Proceedings of the Third International Conference on Innovative Data Systems Research (2007)