Optimizing RDF Data Cubes for Efficient Processing of Analytical Queries

Kim A. Jakobsen, Alex B. Andersen, Katja Hose, and Torben Bach Pedersen

Department of Computer Science, Aalborg University, Denmark
{kah|aban09|khose|tbp}@cs.aau.dk

Abstract. In today's data-driven world, analytical querying, typically based on the data cube concept, is the cornerstone of answering important business questions and making data-driven decisions. Traditionally, the underlying analytical data was mostly internal to the organization and stored in relational data warehouses and data cubes. Today, external data sources are essential for analytics and, as the Semantic Web gains popularity, more and more external sources are available in native RDF. With the recent SPARQL 1.1 standard, performing analytical queries over RDF data sources has finally become feasible. However, unlike their relational counterparts, RDF data cubes stores lack optimizations that enable fast querying. In this paper, we present an approach to optimizing RDF data cubes that is based on three novel cube patterns that optimize RDF data cubes, as well as associated algorithms that transform the RDF data cube. An extensive experimental evaluation shows that the approach allows trading additional storage and/or load times in return for significantly increased query performance. We further provide guidelines for which patterns to apply for specific scenarios and systems.

1 Introduction

Data has become the fundamental resource for making informed decisions in almost every organization. Traditionally, the data used for analytics is mostly internal, e.g., sales, finance, or HR data and is processed using heavy Extract-Transform-Load (ETL) flows and stored in relational data warehouses. It is then analyzed, typically using tools based on multidimensional *data cube* concepts, where important business facts, e.g., a specific lineitem, with associated numerical *measures*, e.g., price or quantity, are organized in a multidimensional (cube) space spanned by *hierarchical dimensions* characterizing the fact, e.g., the product being bought, the customer placing the order, and the order date. Data cubes enable easy and efficient analytical queries that aggregate measure values up to the desired level of detail.

External data sources are becoming more and more important to get the full picture of the situation, often in combination with internal data. Given the growth of the Semantic Web, such external sources become increasingly available in RDF format, e.g., due to efforts in publishing Open Data as Linked Open Data [2]. It is hence desirable to integrate and query external (and internal) data directly in RDF format [9]. Furthermore, the powerful inference of RDF is available to use in the RDF data cubes. RDF data cube vocabularies, such as QB4OLAP [8], which we use in this paper, can be used to specify the desired multidimensional semantics of the RDF data. RDF data can be queried using SPARQL[1] which provides the functionality needed for analytical queries.

[1] http://www.w3.org/TR/sparql11-query/

Popular RDF stores, such as Jena TDB[2] still lack the efficiency of their relational counterparts when answering complex analytical queries. There is thus a significant need for *specialized optimization techniques*.

This paper introduces *cube patterns*, or patterns in short, as specialized optimization techniques of *RDF data cubes*. The patterns are inspired by effective relational representations of data cubes and differ from each other in their level of denormalization. In summary, this paper makes the following novel contributions:

- Proposing three patterns for capturing different levels of denormalization of RDF data cubes: *snowflake pattern*, *star pattern*, and *fully denormalized pattern*.
- Proposing the Semantic Web OLAP Denormalizer (SWOD) transformation algorithm that converts an RDF data cube into a cube in either star pattern or fully denormalized pattern.
- Providing an extensive experimental evaluation, based on the well-known TPC-H dataset [16] transformed into RDF, showing that the proposed patterns allow effective trade-offs between storage space/load times and query performance.

To the best of our knowledge, this is the first paper proposing specialized optimization techniques for RDF data cubes.

The remainder of this paper is structured as follows. In Section 2, we introduce basic concepts of RDF graphs and multidimensional cubes. Then, Section 3 discusses related work. The cube patterns are presented in Section 4. Section 5 explains the transformation algorithms. The results of our experimental evaluation are discussed in Section 6. Section 7 concludes the paper with an outlook to future work.

2 Preliminaries

In this section, we define important concepts and provide a basic understanding of concepts that the rest of the paper is based upon.

RDF Graphs. An RDF graph can be represented as a set of triples of the form (s, p, o) that are each defined by a subject s, a predicate p, and an object o. A triple encodes the existence of a relationship between subject s and object o, the nature of this relationship is described by predicate p.

Given a set of IRIs U, a set of blank nodes B, and a set of literals L, a triple t is defined as $t = (s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$. An RDF graph G consists of a set of triples: $G \subseteq (U \cup B) \times U \times (U \cup B \cup L)$.

An intuitive way of querying a graph is using *triple pattern*s. A triple pattern may contain variables at any position of a triple. Variables are denoted by a leading "?" in their names, e.g., $(?s, p, o)$. A *basic graph pattern* (BGP) consists of a set of triple patterns connected via logical conjunctions.

Given a set of IRIs U, a set of blank nodes B, a set of literals L, and a set of variables V, a triple pattern TP is defined as $TP = (s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. A Basic Graph Pattern BGP is a set of triple patterns: $BGP \subseteq (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. When two or more triple patterns share a variable, we say that there is a join between these triple patterns and the shared variable is the join variable.

When evaluating a BGP, $\{(s_1, p_1, o_1), (s_2, p_2, o_2), \ldots, (s_n, p_n, o_n)\}$ over a graph G we use the notation $G\,(s_1\ p_1\ o_1\ .\ s_2\ p_2\ o_2\ .\ \ldots\ .\ s_n\ p_n\ o_n)$. The result is a bag of

[2] http://jena.apache.org/

bindings for the variables in the BGP. In addition to BGP queries, this paper considers SPARQL 1.1 extensions, such as grouping and aggregation.

Multidimensional Cube. A multidimensional cube is a data structure used for capturing and analyzing data [10,12]; broadly used in Decision Support Systems (DSSs) over relational data. We call a multidimensional cube consisting of RDF an *RDF data cube* or just a cube. We use the QB4OLAP vocabulary [8] to define the structure of cubes, we discuss the alternatives in Section 3; in Section 4 we show how we use QB4OLAP to define the instance data of our cubes.

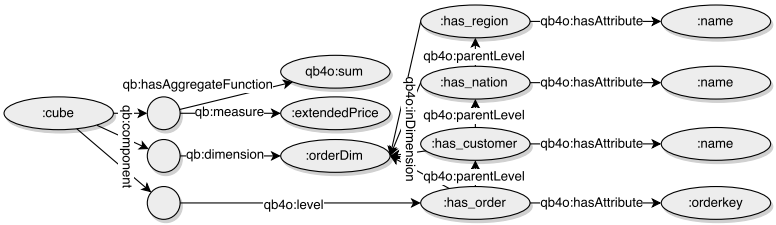


Fig. 1: Example QB4OLAP cube structure

Figure 1 shows a sample of the QB4OLAP cube structure we use in the experiments. Note that some details have been ommitted, e.g., types and non-essential elements. We refer to [8] for a complete definition of QB4OLAP. The cube (`:cube`) has a set of components (blank nodes), which each represent a specific measure, dimension, or level. A cube structure may have an arbitrary number of *measures*; the definition of a measure includes a property used to store a numerical value in the dataset and an aggregation function, e.g. `:extendedPrice` and `qb4o:sum`. At instance level, a set of measure corresponds to a *fact*; it represents the subject of the analysis, e.g., a lineitem representing a single sale. A *dimension*, such as `:orderDim`, characterizes facts and measures by providing context at different levels of detail. Every dimension can have one or more *hierarchies* each with a number of *levels*. The finest granularity of a hierarchy is called the *bottom level*, the coarsest granularity is called the *top level*.

The data in a relational data warehouse is organized using schemas, most commonly known are the *snowflake* and *star schemas*. A *snowflake schema* [10] stores a dimension in several tables where each table contains a level. A snowflake schema is in third normal form, such that data redundancy is minimized. In a *star schema* [10], the dimensions are denormalized, i.e., all levels in a dimension are merged together into one table. When querying a star schema, the number of joins is usually lower because the dimension levels are already joined; this, however, comes at the cost of data redundancy. A third schema is the fully *denormalized schema*, for which all data is denormalized i.e., all levels are merged with the facts. This results in high data redundancy but no joins are needed when querying the denormalized schema.

Whereas optimizing relational data cubes is well-researched, the topic has not yet received much attention in the context of RDF data cubes.

3 Related Work

Decision Support Systems (DSS) have originally emerged from relational database systems but recently started to consider RDF as well. [1], for instance, proposes to use RDF data as so-called situational data, which is data with a short life span. This situational data augments the relational data cubes to form *fusion cubes*. [14] integrates

RDF/OWL ontologies describing domains into relational storage in order to construct multidimensional cubes. Whereas these works focus on ad-hoc data integration using RDF in a relational data warehouse to store and query the data, we examine the possibilities of building a data warehouse based on Semantic Web standards, i.e., RDF and SPARQL 1.1.

There are in general two approaches for constructing RDF data cubes: 1) Explicit cube construction uses a vocabulary to define dimensions, measures, etc. The most commonly used vocabularies to define cubes are: QB [7] and QB4OLAP [8]. The W3C standard QB is used to describe statistical cubes and defines the concepts of facts, dimensions, and levels. The vocabulary QB4OLAP, has been created as an extension of QB, and is more suited for OLAP cubes. It extends QB with aggregate functions, cardinality, and hierarchies. We use QB4OLAP as our main vocabulary because of its expressiveness and because it is specialized for OLAP. 2) Implicit cube construction is when dimensions are implied by the structure of the data. [15] explains how to find measures and dimensions based on a user-defined fact by analyzing an ontology. Similarly, [5] generates an analytical schema which is a graph containing all facts; a fact is selected and an analytical schema instance is created. By inference, a cube with the selected fact can be constructed. Implicit cube construction techniques show much potential but do not support construction of more advanced cubes, such as cubes with multiple aggregation functions or complex hierarchies.

[6] explores the performance of fully denormalized schemas, meaning that all queries are performed on a single relational table – in some cases this improves query evaluation time. For most cases, however, the denormalized schema is outperformed by the snowflake schema. Our experiments show that denormalization improves query time in most cases. In some cases, however, the redundant data cause a to large overhead.

Several approaches addressing OLAP over graph data have been proposed [4,13]. The proposed techniques are highly optimized for graph stores representing data as matrices. [3] relies on an extension of SPARQL, uses attribute graphs, and a special framework. Likewise, [17] outlines a system featuring a special storage architecture and components optimized to support analytical queries. If the queries are known beforehand, then further optimization is possible by pre-computing results in the form of materialized cubes [11]. The techniques we propose in this paper, however, build upon existing standards, efficiently support arbitrary queries, and can be implemented in any triple store – the only requirement being that SPARQL 1.1 is supported.

4 Cube Patterns

In this section, we present three patterns for RDF data cubes: *snowflake pattern*, *star pattern*, and *fully denormalized pattern*. We will discuss the relationship between the different patterns and how to derive them in Section 5.

Snowflake Pattern. An RDF data cube in snowflake pattern is normalized such that each level is represented by one class in the ontology; an instance of a level is called a *level member*. Levels are arranged in hierarchies that are organized in dimensions. Facts

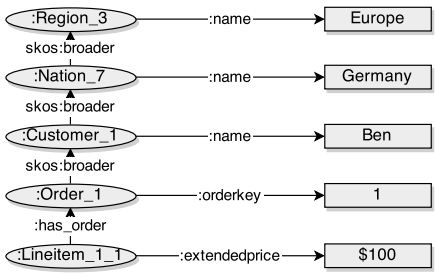


Fig. 2: Snowflake pattern example

link to level members of the bottom level of a dimension. Figure 2 illustrates the fact representing Ben's purchase, `:Lineitem_1_1`, linking to a level member at the Order level, `:Order_1`, which links to its parent at the Customer level member, `:Customer_1`, and so on. In this example, there is a single attribute describing each level member; in practice, more attributes are often used.

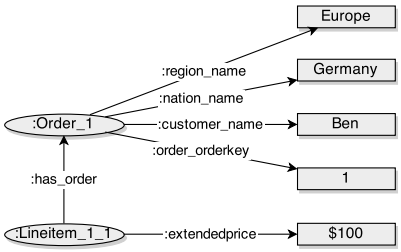


Fig. 3: Star pattern example

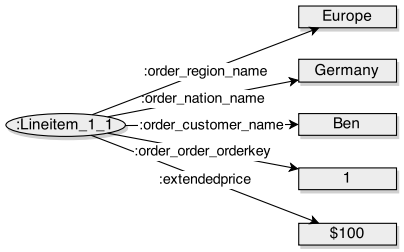


Fig. 4: Denormalized pattern example

Star Pattern. In a star pattern RDF data cube, all levels have been denormalized into a dimension. An instance of a dimension is called a *dimension member*, e.g., `:Order_1`. A dimension member is a resource that has *all* attributes of *all* levels in the given dimension. Figure 3 shows the Order dimension and a fact relating to Ben's purchase modeled as star pattern. In Figure 3, the level members `:Region_3`, `:Nation_7`, `:Customer_1`, and `:Order_1` from Figure 2 are merged into a dimension member, called `:Order_1`.

Fully Denormalized Pattern. In the fully denormalized pattern, in short denormalized pattern, all level attributes are attached directly to the facts. In Figure 4 we see Ben's purchase modeled as denormalized pattern. The level members `:Region_3`, `:Nation_7`, `:Customer_1`, and `:Order_1` from Figure 2 are directly connected to the fact `:Lineitem_1_1`.

5 Denormalization Algorithms

In this section, we discuss how to convert an RDF data cube from snowflake pattern into star pattern and denormalized pattern by using the SWOD (Semantic Web OLAP Denormalizer) algorithm. For ease of presentation, we first present the conversion from snowflake pattern into star pattern and afterwards we highlight the differences for the denormalized pattern. We first explain `SwodStar` (Algorithm 1), which is the main algorithm and then proceed with its auxiliary functions. We assume that a cube is in snowflake pattern because the commonly used multidimensional vocabularies encourage this, and it is arguably the most intuitive way of organizing an RDF data cube. We assume that the give QB4OLAP ontology strictly describe a single cube.

Notation. We use two kinds of variables in the algorithms: BGP variables, which are prefixed with a question mark "?", and regular algorithm variables. In some places, we use an algorithm variable in a BGP, which represents a constant (URI, literal, or blank node) in the BGP. A graph G' can be constructed by evaluating a BGP on a graph G e.g., $G' = \{(s, p, o) \in G\,(?s$ rdf:type $t\ .\ ?s\ ?p\ ?o)\}$. G' contains all triples with the subject s that are instances of class t in graph G. If a BGP does not contain variables, the result of its evaluation is a boolean value indicating whether the represented set of triples is present in a given graph. We say that we *merge* a triple a into another triple b

in a graph G, when a triple is created with the subject of b and the predicate and object of a. Note that we use [] to represent a blank node.

```
1  Function SwodStar(cube,onto) is
2      cube′ = ∅;
3      bottomLevels = GetBottemLevels();
4      foreach (level, dim) ∈ bottomLevels do
5          cube′ = cube′ ∪ MergeLevel(cube,onto,level,dim);
6          cube′ = cube′ ∪ {(fact, level, levelMem)|(fact, levelMem) ∈ cube(?fact qb:dataSet [].
           ?fact level ?levelMem)};
7      end
8      foreach (fact, prop, obj) ∈ cube(?fact qb:dataSet [] . ?fact ?prop ?obj) do
9          if onto(?structure qb:component ?component. ?component qb:attribute prop.) ≠ ∅ ∨
           onto(?structure qb:component ?component. ?component qb:measure prop.) ≠ ∅ then
10             cube′ = cube′ ∪ {(fact, prop, obj)};
11         end
12     end
13     return cube′;
14 end
```

Alg. 1: Conversion of an RDF data cube in snowflake pattern to star pattern

Cube Denormalization. `SwodStar` (Algorithm 1) converts an RDF data cube from snowflake pattern into star pattern. It has two input parameters: the RDF data cube in snowflake pattern (cube), see Figure 2, and the ontology describing the QB4OLAP cube (onto), see Figure 1. `SwodStar` outputs an RDF data cube (cube′) in star pattern, see Figure 3.

The function `GetBottemLevels` finds the bottom level for each dimension and returns a set of pairs (level, dim) (line 3). This is possible by traversing the levels (`qb4o:level`) defined in the QB4OLAP ontology and finding the levels that are not a parent level (`qb4o:parentLevel`), see Figure 1. The resulting set of pairs is used as input to `MergeLevel` (Algorithm 2) along with cube and onto. The `MergeLevel` algorithm (Algorithm 2) is called (line 5) to recursively merge level members with their ancestors into dimension members, the obtained triples are added to the new snowflake pattern cube cube′. Next, `SwodStar` connects the dimension members to the facts (line 6), see Figure 3; the algorithm creates triples such as (`:Lineitem_1_1`, `:has_order`, `:Order_1`). These triples are inserted into the star pattern cube (cube′). The next step is to loop through the fact triples in the snowflake pattern cube and add them to cube′ (lines 8–12). In our example, we only have a single fact `:Lineitem_1_1`, which has a single measure, namely `:extendedprice`; we add this triple to cube′. Finally, we have converted the RDF data cube from snowflake pattern into star pattern.

Unbalanced Hierarchies. As dimension hierarchies may be unbalanced [10], it is important to handle such cases. In Figure 5, we see an abstract example of an unbalanced hierarchy. The circles represent level members, which have arrows labeled with numbers representing the number of their *attribute sets*. An attribute set is the set of attributes of a level member.

When denormalizing, level members are merged into the bottom level member. In the case where a level member does not have any children but is not the bottom level of the dimension, then the ancestor level members are merged into it. For example the Customer (cust′) does not have any orders and the Nation (nation) and Region (region) level members are merged into the Customer (cust′) along with their attributes (Figure 6). In the denormalized pattern, the same principle is used (Figure 7).

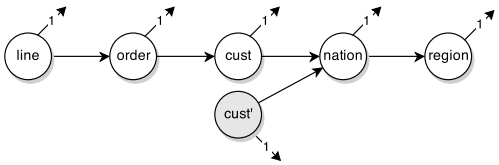


Fig. 5: Snowflake pattern

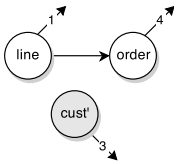


Fig. 6: Star pattern

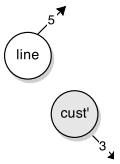


Fig. 7: Denormalized pattern

```
1  Function MergeLevel(cube,onto,level,dim) is
2      dimMembers = ∅;
3      foreach parLevel ∈ onto(level qb4o:parentLevel ?parLevel . ?parLevel qb:inDimension dim) do
4          parInsts = MergeLevel(cube,onto,parLevel,dim);
5          foreach parInst ∈ parInsts(?parInst qb4o:inLevel [ ]) do
6              foreach levelMem ∈ cube(?levelMem skos:broader parInst) do
7                  dimMembers = dimMembers ∪ {(levelMem, prop, obj)|(prop, obj) ∈
                   parInsts(parInst ?prop ?obj)};
8              end
9              if cube(?levelMem skos:broader parInst) = ∅ then
10                 dimMembers = dimMembers ∪ {(parInst, prop, obj) ∈
                   parInsts(parInst ?prop ?obj)};
11             end
12         end
13     end
14     foreach prop ∈ onto(level qb4o:hasAttribute ?prop) do
15         dimMembers = dimMembers ∪ {(levelMem, MergeURIs(level,prop) obj) ∈
           cube(?levelMem qb4o:inLevel level. ?levelMem prop ?obj)};
16     end
17     dimMembers = dimMembers ∪ {(levelMem, qb4o:inLevel, level)|levelMem ∈
       cube(?levelMem qb4o:inLevel level)};
18     return dimMembers;
19 end
```

Alg. 2: Generation of level members through recursively merging parent levels

Merging Level Members. The `MergeLevel` function (Algorithm 2) recursively merges level members in a top down manner into dimension members. It has four input parameters: a RDF data cube (cube), a QB4OLAP ontology (onto), the current level (level) and the dimension of the level (dim). If the level member (level) has any parent levels, then we recursively call `MergeLevel` for each of the parent levels (line 4). When a top level member is reached then no parent levels will exist, thus line 5–12 will be skipped. dimMembers contains all attributes of the ancestor level members, we merge the attributes of the current level member with these (line 15). The function `MergeURIs` creates new attributes to avoid ambiguity among properties used in multiple levels. We add the level name as a prefix to the property e.g., `:name` becomes `:nation_name`. We create mapping between the old and the new properties, so that we can always rewrite queries to match a given triple pattern. The final step adds a triple with the name of the current level (line 17).

When we return from the recursion in line 4, we loop through every instance of the parent levels (line 5). We merge the parent level members with the current level member (line 7). To handle unbalanced hierarchies, we add ancestor level members that do not have any children in the current level (lines 9–11).Every subsequent return is handled in the same way. The returned triples of the initial call are dimension members, rather than level members as illustrated in Figure 3.

Fully Denormalized Cube. To construct a denormalized pattern cube, we use similar algorithms as for the star pattern cube. In addition to creating the dimension members, we further merge them with the facts, see Figure 4. To avoid ambiguity among

properties used in multiple dimensions, we additionally add the dimension name as a prefix to the property e.g., `:order_nation_name`. In case of unbalanced hierarchies, e.g., a customer without any orders, we create dimension members that are not connected to any facts, as oppose to level members.

Query Rewriting. To enable transparency for the user, formulating a query in snowflake pattern, we need to rewrite the query such that it matches the structure of the data in star pattern or denormalized pattern. In principle, a query formulated on the RDF data cube in snowflake pattern can be transformed in a similar way as the cubes. More precise, we loop through the triple patterns in the original query and determine the referenced levels and dimensions using the QB4OLAP ontology. If the subject in the triple pattern does not correspond to a bottom level, then we merge it with the corresponding bottom level in the dimension and change the predicate to match the naming scheme, e.g., `?nation :name ?name` becomes `?order :nation_name ?name`. Recall that level members are connected with the `skos:broader` predicate, see Figure 2. When we denormalize the cube the level members are merged, thus triple patterns with the `skos:broader` predicate can be removed. Query 1.1 is in snowflake pattern and finds how much money customers from all nations have spent. Query 1.2 shows the same query rewritten to match the star pattern.

```
SELECT ?name sum(?price)
WHERE {
  ?lineitem :extendedprice ?price ;
            :has_order ?order .
  ?order skos:broader ?customer .
  ?customer skos:broader ?nation .
  ?nation :name ?name .
}
GROUP BY ?name
```

Query 1.1: Snowflake pattern query

```
SELECT ?name sum(?price)
WHERE {
  ?lineitem :extendedprice ?price ;
            :has_order ?order .
  ?order :nation_name ?name .
}
GROUP BY ?name
```

Query 1.2: Star pattern query

In some cases query rewriting is not trivial. Assume, we are looking for the number of orders placed by the customer whose name is Ben. The triple `:Customer_1 :name "Ben"` would be a match in the snowflake pattern (see Figure 2) and we can use the triple pattern `?order skos:broader :Customer_1` to find his orders. For the star pattern, however, we would find several triples, such as `:Order_1 :name "Ben"` and `:Order_2 :customer_name "Ben"`. Here we do not know if Ben and Ben is the same person. When the object is a literal (e.g. Ben) and the predicate `:name` has not been defined as a functional property, we cannot know if it is the same customer or not. In this case, we have to augment the query with a triple pattern using the functional predicate of the customer and add a group by statement e.g., `GROUP BY ?customer_ID`. If no functional property exist then we use the URI of the customer resource to group the customers uniquely.

6 Experiments

In this section, we present the results of evaluating the techniques and report on the performance of generating, loading, and querying the three patterns using data sets of different sizes.

Hardware platform. The experiments were run on a HP ProLiant DL385 server with an AMD Opteron(tm) processor 6376 with 32 cores, it has 256 GB DDR3 RAM and is running Ubuntu 14.04.1 LTS (Trusty Tahr). The data was stored on a 1 TB SCSI disks running in a HP Smart Array.

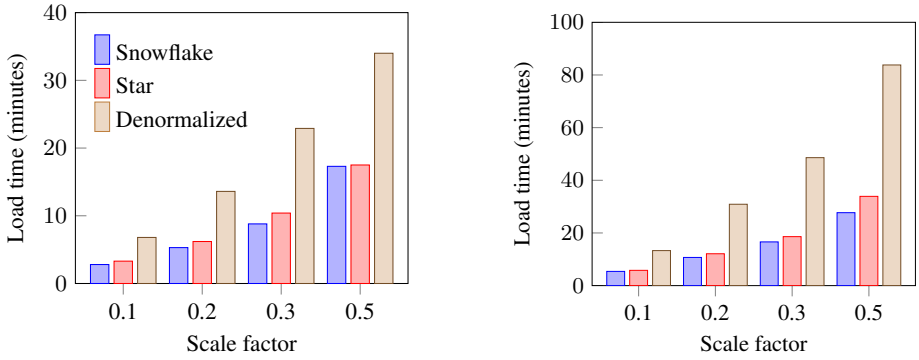
Dataset and queries. We use the well-known TPC-H benchmark [16], which was designed for relational data warehouses and comes with a CSV data generator as well as with 22 complex and challenging analytical queries.

The TPC-H benchmark data is structured data about sales modeled as a snowflake schema. The analytical queries in the benchmark are originally provided as SQL query templates. We created SPARQL query templates based on the provided SQL benchmark query templates; we transformed the query template such that they match the star pattern and denormalized pattern, hence we have three sets of 22 query templates. The query templates and further details are available at our homepage[3].

Pattern generation. We generated the TPC-H dataset at different scale factors using the supplied DBGEN program[4]. We defined cubes by converting the generated comma separated files using the csv2ttl program, which is a part of the BIBM project, and manually defined a snowflake pattern cube using QB4OLAP. This cube is loaded into Virtuoso-opensource 7.10, where we run the SWOD algorithm, as a result we obtain cubes in star pattern and denormalized pattern. Table 1 shows the sizes of the test data sets in millions of triples The star pattern (denormalized pattern) cubes are about 16% (173%) bigger in terms of the number of triples than the snowflake pattern cubes.

	Scale factor			
	0.1	0.2	0.3	0.5
Snowflake	15.2	30.4	45.5	75.9
Star	17.7	35.4	53.0	88.4
Denormalized	41.5	82.9	124.5	207.3

Table 1: Dataset sizes (millions of triples)



(a) Virtuoso (b) Jena

Fig. 8: Dataset load times

Load times. We loaded the three cubes into two different stores: we use Apache Jena 2.12 TDB and Virtuoso-opensource 7.10. Figures 8(a) and 8(b) report the times for loading the generated cubes. Virtuoso uses on average 12.5% more time to load the star pattern cube than the snowflake pattern cube and Jena uses 14.2% more time. The denormalized pattern cube takes on average 138.3% longer in Virtuoso, while it takes Jena 183% longer, again compared to the snowflake pattern cube. The increased load time is due to the increased number of triples.

Query evaluation. Each of the three sets of the 22 queries are executed on the two triple stores (Jena TDB and Virtuoso). The queries were run in random order with

[3] http://extbi.cs.aau.dk/swod

[4] http://sourceforge.net/projects/bibm/

	Scale factor 0.2						Scale factor 0.5					
	Virtuoso			Jena			Virtuoso			Jena		
	Snow	Star	Denorm	Snow	Star	Denorm	Snow	Star	Denorm	Snow	Star	Denorm
Q1	7.3	**7.2**	7.3	**71.8**	78.4	76.3	15.0	14.9	**14.8**	**180.9**	193.4	184.6
Q2	1.7	1.9	**1.1**	**5.7**	8.0	54.5	**2.1**	2.9	2.4	**31.5**	34.6	244.5
Q3	2.9	**0.4**	1.3	30.0	33.0	**25.9**	6.2	**0.8**	2.7	77.0	78.3	**62.2**
Q4	8.8	8.3	**3.0**	33.57	35.7	**31.9**	22.7	20.1	**7.4**	**86.9**	125.1	135
Q5	4.3	1.3	**0.2**	38.6	37.1	**23.0**	7.8	3.3	**0.4**	97.6	86.2	**56.6**
Q6	**1.8**	**1.8**	**1.8**	**19.6**	20.8	21.1	2.7	2.8	**2.6**	**47.9**	51.2	51.0
Q7	7.8	1.8	**0.7**	35.5	31.6	**20.5**	19.9	4.4	**1.6**	90.2	77.6	**49.7**
Q8	2.5	0.2	**0.1**	51.8	36.8	**25.0**	7.1	0.4	**0.2**	131.4	90.7	**59.9**
Q9	31	**6.0**	8.4	65.3	47.0	**33.4**	82	**16.9**	23.6	167.9	110.4	**78.7**
Q10	–	**1.2**	2.2	15.8	15.2	**13.5**	–	**1.9**	2.8	39.2	37.1	**31.7**
Q11	1.2	**0.3**	0.7	**0.6**	**0.6**	2.3	1.7	**0.5**	1.2	**1.2**	1.3	5.5
Q12	**1.6**	**1.6**	2.6	26.3	27.2	**23.5**	**3.0**	3.1	5.8	63.6	66.4	**54.2**
Q13	**1.6**	1.7	7.3	**8.4**	9.7	32.7	**4.8**	5.1	19.7	–	–	–
Q14	2.3	8.4	**1.1**	**21.7**	24.0	23.5	5	25.3	**2.0**	**53.8**	57.0	55.4
Q15	3.2	19.4	**2.9**	**31.1**	377.7	–	**5.9**	55	6.3	**106.9**	–	–
Q16	1.8	**1.7**	4.4	**2.1**	5.0	34.9	4.5	**4.1**	11.2	**5.2**	11.8	81.1
Q17	3.6	**0.1**	**0.1**	91.8	**40**	54.2	6.2	**0.2**	**0.2**	–	–	–
Q18	2.0	**1.9**	2.0	**18.4**	19.9	19.9	4.4	**3.9**	4.2	228.8	206.2	**169.7**
Q19	4.1	2.9	**1.5**	**21.4**	27.7	21.8	8.1	6.5	**2.7**	52.3	64.0	**52.1**
Q20	3.1	**1.6**	11.2	–	–	–	6.2	**3.9**	28.6	–	–	–
Q21	9.3	2.2	**0.6**	43.8	39.4	**29.2**	21.5	5.1	**1.2**	107.7	93.5	**71.0**
Q22	**0.2**	1.7	2.6	**3.6**	10.5	42.6	**0.4**	3.5	5.7	**3.7**	28.4	103.0
Avg.	50.1	3.3	**2.9**	**74.4**	87.5	118.6	56.2	8.4	**6.7**	**207.9**	246.1	252.1
G.M.	3.9	1.8	**1.5**	**22.6**	27	35.9	7.8	3.8	**3.2**	**74.4**	91.3	109.3
Count	4	11	11	11	2	9	5	8	10	9	0	10

Table 2: Query evaluation (seconds)

random parameters as specified by the TPC-H manual [16]. We ran all query sets 12 times, removed the slowest and the fastest run, and took the average of the remaining 10.

Table 2 shows the detailed query runtimes for scale factors 0.2 and 0.5. Due to space limitations and because the observations for these results also hold for the other scale factors, we omit detailed runtimes for the remaining ones. Some queries time out and are marked with "–", symbolizing that the query time exceeds 1000 seconds. We calculated the minimum average and minimum geometric mean (G.M.), we refer to this as minimum because queries that time out are contributing to these numbers with a runtime of 1000 seconds but in reality exceed 1000 seconds. For each query, pattern, and store we mark the fastest execution.

As we can see in Table 2, Virtuoso shows a low average and geometric mean for the denormalized pattern in both scale factors. On average, in Virtuoso, the star pattern (denormalized pattern) is 6 (8) times faster than the snowflake pattern, which is particularly interesting because the snowflake pattern is the pattern in which most RDF data cubes are available. In Virtuoso at scale factor 0.5, 11 out of the 22 queries are fastest on either the star pattern or the denormalized pattern – and only 4 on the snowflake pattern. This shows that Virtuoso is able to handle the increased amount of data and benefits from the denormalization.

For Jena, the results are quite different; the snowflake pattern is the fastest in terms of average/geometric mean. We see that the snowflake pattern cube in scale factor 0.2

has 11 of the fastest query times and in scale factor 0.5 it has nine. The TDB database in Jena is, unlike Virtuoso, experiencing difficulties with high numbers of triples, thus several queries times out, especially when the cubes are denormalized.

For some queries, the two stores perform best on different patterns for the same query. This is caused by the differences in storage, caching, and optimization techniques of the two triple stores. In Virtuoso, query 12 is fastest on the snowflake pattern and star pattern. In Jena, however, the denormalized pattern is fastest. Query 15 contains three subqueries and is not able to execute in denormalized pattern in Jena and in scale factor 0.5 star pattern also exceeds the timeout limit; this illustrates how big a difference the query optimizer makes.

When comparing the results of the two stores for scale factors 0.2 and 0.5, we see only few changes regarding which pattern is fastest on the different queries. For most of the queries where this is changing, e.g., queries 1, 6, and 18, the runtimes for all patterns are very close. Query 6 does not query any levels but only facts, this means that only subject-subject joins are made. The runtimes for the snowflake pattern, the star pattern, and the denormalized pattern are relatively similar in both stores. This indicates that queries of this particular type do not benefit from denormalization.

For some queries, both Jena and Virtuoso performs best on the same pattern. Query 5, for instance, involves seven different levels, this entails many subject-object joins, which resemble long path queries in the snowflake pattern and the star pattern but only a short path in the denormalized pattern. Therefore, query 5 is evaluated fastest on the denormalized pattern.

In general, we observe the tendency that queries that span many distinct levels benefit from denormalization while the opposite is also true: queries that span only a few levels perform better on the snowflake pattern.

Conclusion of the experiments. Based on the three measures load time, storage size, and query time we have gained a deeper understanding of when to use the three patterns. We see that the denormalized pattern has more than twice as many triples as the other patterns and much larger load time but shows the most potential in terms of query time for Virtuoso, for which the denormalized pattern performs up to 35 times faster (in scale factor 0.5). In Virtuoso the star pattern cube represents a sweet spot between query time, load time, and amount of triples. Jena does not in general gain a performance increase in terms of query time when using denormalization.

The fully denormalized pattern is highly recommended for Virtuoso when dealing with static datasets – for frequently changing data, the update costs are too high. This pattern also requires a hardware platform with sufficient storage due to the introduced data redundancy and an underlying system that can efficiently handle a high number of triples. Alternatively, we recommend using the star pattern because of the modest increase in triples and load time – as compared to the snowflake pattern– while on average yielding more than 6 times faster query times in Virtuoso. Based on our experimental results, we do not recommend denormalization in Jena.

7 Conclusion

Motivated by the increasing need to store and query analytical data in RDF format, this paper presented query optimization techniques to increase the performance of queries on RDF data cubes. Inspired by relational data cube representations, the paper proposed three novel cube patterns for RDF data cubes: snowflake pattern, star pattern, and fully

denormalized pattern. Furthermore, this paper presented the Semantic Web OLAP Denormalizer (SWOD) algorithm that transforms the pattern of the RDF instance data. Finally, the paper provided an extensive experimental evaluation, based on an RDF version of the TPC-H benchmark, examining storage space, load time, and query time. The evaluation showed that the proposed patterns are effective for improving query performance at the expense of additional storage space and that their performance also depends on the underlying triple store. Interesting directions for future work include implementing additional optimizations for data cube processing, such as materialized views, as well as the integration of the proposed approach directly within a triple store.

Acknowledgment

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement No. DFF-4093-00301.

References

1. A. Abelló, J. Darmont, L. Etcheverry, and et al. Fusion Cubes: Towards Self-Service Business Intelligence. *IJDWM*, 9(2), 2013.
2. A. B. Andersen, N. Gür, K. Hose, K. A. Jakobsen, and T. B. Pedersen. Publishing Danish Agricultural Government Data as Semantic Web Data. In *JIST*, pages 178–186, 2014.
3. S.-M.-R. Beheshti, B. Benatallah, H. Motahari-Nezhad, and M. Allahbakhsh. A Framework and a Language for On-Line Analytical Processing on Graphs. In *WISE'12*, pages 213–227.
4. C. Chen and et al. Graph OLAP: a multi-dimensional framework for graph data analysis. *Knowledge and Information Systems*, 21(1), 2009.
5. D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatis. RDF analytics: lenses over semantic graphs. In *WWW*, 2014.
6. J. a. P. Costa, J. Cecílio, P. Martins, and P. Furtado. ONE: A Predictable and Scalable DW Model. In *DaWaK*, 2011.
7. R. Cyganiak and D. Reynolds. The RDF Data Cube Vocabulary. http://www.w3.org/TR/2014/REC-vocab-data-cube-20140116/.
8. L. Etcheverry and A. A. Vaisman. QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In *COLD*, 2012.
9. D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi. Processing Aggregate Queries in a Federation of SPARQL Endpoints. In *ESWC*, pages 269–285, 2015.
10. C. S. Jensen, T. B. Pedersen, and C. Thomsen. *Multidimensional Databases and Data Warehousing*. Morgan & Claypool Publishers, 2010.
11. B. Kämpgen and A. Harth. No Size Fits All – Running the Star Schema Benchmark with SPARQL and RDF Aggregate Views. In *ESWC*, pages 290–304, 2013.
12. R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, 2. edition, 2002.
13. N. Li, Z. Guan, L. Ren, J. Wu, J. Han, and X. Yan. Iceberg: Towards iceberg analysis in large graphs. In *ICDE*, pages 1021–1032, 2013.
14. M. Niinimäki and T. Niemi. An ETL Process for OLAP Using RDF/OWL Ontologies. *Journal on Data Semantics*, 2009.
15. O. Romero and A. Abelló. Open Access Semantic Aware Business Intelligence. In *eBISS*, pages 121–149, 2013.
16. Transaction Processing Performance Council (TPC). *TPC BENCHMARK™H (Decision Support) Standard Specification*, revision 2.16.0 edition, 2013.
17. M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux. dipLODocus[RDF] – Short and Long-Tail RDF Analytics for Massive Webs of Data. In *ISWC*, 2011.