A methodological approach to incremental view maintenance for optimizing SPARQL queries in Solid

Dore Staquet[1,2,*,†], Bart Buelens[2,†] and Jan Van den Bussche[1,†]

[1]Data Science Institute, Hasselt University
[2]Flemish Institute for Technological Research (VITO)

Abstract

This contribution explores innovative strategies for efficient information retrieval in large-scale Solid deployments through the development of incremental view maintenance techniques. Assuming a pivotal role for web agents and aggregators in managing SPARQL queries within decentralized data architectures, we propose the adaptation of established relational database methodologies, specifically the counting algorithm and propagation rules, for the maintenance of materialized views in RDF (Resource Description Framework) data settings. Our research critically assesses the applicability of these algorithms to RDF, addressing the unique challenges posed by SPARQL and linked data's graph nature. We demonstrate the incremental maintainability of aggregation functions like COUNT, SUM, and AVG, while highlighting the limitations for functions such as MIN, MAX, and SAMPLE. By formulating these methodologies using SPARQL algebra, we set the stage for practical implementations that significantly enhance query response times without necessitating full data re-computation. This approach not only underscores the feasibility of applying relational database concepts to linked data but also sets a foundational framework for future research aimed at optimizing data retrieval processes in Solid-based applications and ecosystems at web-scale.

Keywords

RDF, linked data querying, materialized view

1. Introduction

In large-scale Solid [1, 2] deployments, efficient information retrieval will be crucial. We adopt the model of web agents, which can be seen as decentralized data processing entities [3]. One important capability of such agents is responding to queries submitted to Solid pods. Queries can be submitted by partners – parties engaging directly with solid pods, or by aggregators – intermediary service providers that can be accessed by other aggregators or partners [4]. SPARQL queries drive the information retrieval process in this decentralized data architecture. In our research, we envisage both web agents and aggregators to maintain materialized views to swiftly respond to SPARQL queries. Figure 1 provides a schematic overview of our setting, in this case with three pods and agents providing their data to several agents and one aggregator. In database design and operations generally, maintaining materialized views on the underlying

The 1st Solid Symposium Poster Session, co-located with the 2nd Solid Symposium, May 02 – 03, 2024, Leuven, Belgium
*Corresponding author.
†These authors contributed equally.
dore.staquet@uhasselt.be (D. Staquet); bart.buelens@vito.be (B. Buelens); jan.vandenbussche@uhasselt.be (J. V. d. Bussche)
0000-0002-6115-6901 (D. Staquet); 0000-0001-7734-3747 (B. Buelens); 0000-0003-0072-3252 (J. V. d. Bussche)

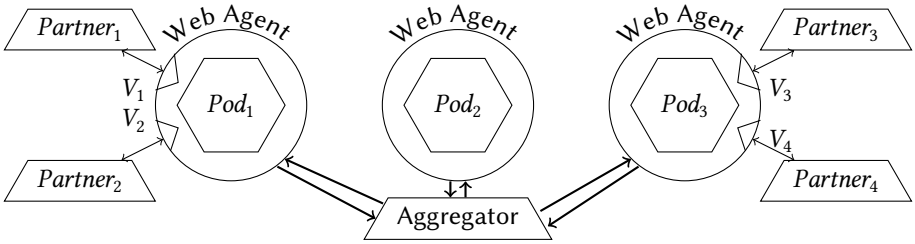


Figure 1: Three pods together with their web agents which handle which trusted partners, such as the aggregator in the illustration, can access which parts of the data. Web Agents provide views V_i which trusted partners can utilize.

data is a key facilitator of efficient information retrieval, certainly so in decentralized models. Materialized views allow to efficiently respond to repeated querying. Often, identical queries are repeated over time to take modifications of the underlying data into account. View maintenance is the approach to update views following changes in the underlying data. A baseline strategy to update a view following a change in the data is to re-execute the query and recompute the view from scratch. In this work, we develop strategies to do this incrementally, avoiding full re-computation where possible, by using the changes in the data only as opposed to the full data set needed to compute the view. Our approach to incremental view maintenance in a Solid setting is based on theory and algorithms established for relational database management systems, which we will apply to linked data [5] in RDF (Resource Description Framework), the data standard used in Solid [6].

2. Related work

Incremental view maintenance has a history in the relational setting, where it has been studied extensively [7, 8, 9]. To extend on this in regards to SPARQL, where the LeftJoin (OPTIONAL), is very important, there has also been research on outer joins in incremental view maintenance. Larson and Zhou propose an alternative [10] to Qian and Wiederhold's [8] propagation rules. Also Griffin and Kumar expand propagation rules for outer join and semijoin [11]. Regarding multiset semantics, there was a proposal which include duplicate elimination by Griffin and Libkin [12]. DBToaster, a state-of-the-art system in the relational setting for incremental view maintenance, also works with multiset semantics [13]. There has been published work on incremental maintenance on linkset views in SPARQL [14], but to the best of our knowledge not yet on incremental view maintenance for general SPARQL patterns.

3. Methods

We revisit methods known from the relational database literature and study how they can be modified and adapted to our needs, to operate on RDF data.

We envisage these methods to be implemented as capabilities of web agents, as illustrated in Figure 1. The web agents in our setup run server-side close to the RDF data, which is stored in Solid pods. A key capability of such a web agent is exposing a SPARQL endpoint, which

is not part of the Solid specification. The web agent acts as a layer on top of the Solid pod, maintaining views on the data inside the pod, providing functionalities to set up new views for clients. Aggregators too are server side components that maintain aggregated views integrating data from multiple pods.

First, we consider the counting algorithm [15] and its applicability in a SPARQL and linked data setting. The counting algorithm is, in the relational setting, one of the most general solutions to incremental view maintenance, i.e., updating the view using only the data changes and not recomputing the entire view. The algorithm works by keeping track of the count of tuples – or triples, for RDF data – that contribute to each value in the view. The algorithm supports all of relational algebra, plus aggregation for incrementally computable aggregate functions. Extending this algorithm from relational databases, what it was designed for, to RDF requires handling forms of complexity that are characteristic of SPARQL and of the graph nature of the data. This is a topic of our ongoing research.

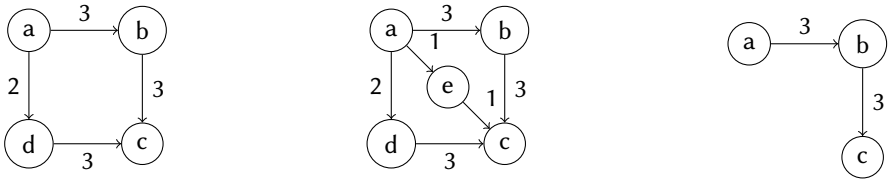
We formulate the incremental view maintenance issue as follows. Let D be an RDF dataset, and Q a SPARQL query. An update to the dataset is written as ΔD. The original view based on the query Q, $V_Q(D)$, is to be recomputed taking the change into account, giving rise to an updated view $V_Q(D, \Delta D)$. Incremental view maintenance aims to obtain an efficiently computable ΔV_Q such that $V_Q(D, \Delta D) = V_Q(D) + \Delta V_Q(\Delta D)$, avoiding the need to access the full data D again to update the view.

4. Results

Before detailing algorithm specifics, we briefly reflect on the incremental maintainability of certain views in the first place. In particular, we study the aggregator functions commonly used in SPARQL, and note that the aggregation functions COUNT, SUM and AVG are incrementally maintainable, while MIN, MAX and SAMPLE are not, at least not always. For example MAX: when the query Q selects the maximum of some values in D, $V_Q(D)$ is the maximum value; if ΔD entails the deletion of any value other than the maximum, the new maximum is equal to the old one; however when the first maximum is deleted, a new one needs to be computed – for which the complete data set is needed again. We provide proofs or counterexamples of the common operators.

For example, based on Gupta and Mumick's work[15], an aggregation with the rule $M(x, y, m) \leftarrow GROUPBY(B(x, y, c), [x, y], m = MIN(c))$ where rule B is defined as $B(x, y, m_1 + m_2) \leftarrow G(x, z, m_1), G(z, y, m_2)$. As illustrated in Figure 2, Graph G showcases an issue that encounters when deleting a node with a minimum aggregation in the view maintenance. In Figure 2a Graph G has $M = ((a, c, 5))$ as a result. When adding a node, as seen in Figure 2b, the minimum is maintained incrementally as it should only compute a new minimum when the edges relating to the new addition equate to the new minimum. In Figure 2b $M = ((a, c, 2))$. However when deleting a node, a problem occurs where the view can no longer be incrementally maintained. Figure 2c shows that when node d gets deleted, there is no way of knowing what the new minimum should be without having to re-evaluate the entire graph from scratch. The result in Figure 2c is $M = ((a, c, 6))$, but this can only be computed by recomputation.

To investigate the algorithms mentioned in the previous section, we express SPARQL queries



(a) Graph G. (b) Addition of node *e*. (c) Deletion of node *d*.

Figure 2: Graph G in three different scenarios, (a) shows the base graph, (b) an addition of a node *e* to graph G and (c) a deletion of node *d* from graph G.

in formal SPARQL algebra, allowing formal derivations and proofs, and serving as instructions when implementing SPARQL query plans in practice.

A foundational algorithm for incremental view maintenance in relational database systems is the counting algorithm [15]. We adapt this algorithm to work for RDF data. Now, counts of solution mappings must be kept and the required updates to views specified. We address a number of issues. An important one, for example, is the handling of NULL values – a common concept in relational databases. In RDF, or linked data more generally, NULL is not stated, rather, that specific triple is simply not present. In SPARQL, selecting such data can be achieved using the OPTIONAL function.

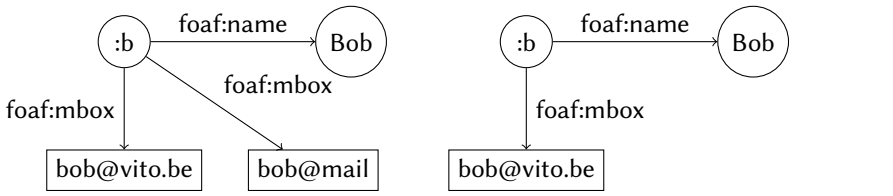
Propagation rules [8, 9] offer a fairly straightforward optimization recipe for incremental view maintenance, once the rules are established and proven correct. We address this latter aspect for RDF data by revisiting the rules for relational data and rewriting them for RDF. This includes rules for differences, unions, outerjoins, semijoins, etc.

For example, illustrating an incremental view maintenance rule on a SPARQL context, assume a Basic Graph Pattern rule where G is the graph on which the query is computated. This rule could look like $B(x, y) \leftarrow G(x, \text{foaf:name}, y), G(x, \text{foaf:mbox}, z)$. Furthermore ΔG represents a changes made to the base graph. Following the definitions of the delta rules or counting algorithm, the delta ruling for ΔB becomes

$$\Delta B(x, y) \leftarrow \Delta G(x, \text{foaf:name}, y), G(x, \text{foaf:mbox}, z)$$

$$\Delta B(x, y) \leftarrow G'(x, \text{foaf:name}, y), \Delta G(x, \text{foaf:mbox}, z)$$

Here, $G' = G + \Delta G$ denotes the updated graph. Now $B' = B + \Delta B$ achieves the desired new relation by calculation of the changes to the view. To visualise this, take a graph shown in Figure 3a where there exists a graph $G((\text{:b}, \text{foaf:name}, \text{Bob}), (\text{:b}, \text{foaf:mbox}, \text{bob@mail}), (\text{:b}, \text{foaf:mbox}, \text{bob@vito.be}))$. For this graph we see that $B((\text{:b}, \text{Bob}) * 2)$ within the counting algorithm, derived twice here as Bob possesses two mailboxes. Removing one of the mailboxes generates the change to the graph as $\Delta G((\text{:b}, \text{foaf:mbox}, \text{bob@mail}) * -1)$ indicating that this triple gets deleted once from the base graph. By executing the counting algorithm ΔB becomes $((\text{:b}, \text{Bob}) * -1)$, making $B' = ((\text{:b}, \text{Bob}) * 1)$. By Figure 3b it is evident that this is correct.



(a) Graph G. (b) ΔG deletes (:b, foaf:mbox, bob@mail).

Figure 3: Graph G (a) before and (b) after applying ΔG.

5. Discussion

In the current research we are well underway of using elements of the counting algorithm and propagation rules to be transferred to a linked data setting to render them useful for our envisaged approach in a Solid model as sketched in Figure 1. Formulating these algorithms in SPARQL algebra terms will allow us to move to a general implementation quickly.

In the near future we will benchmark our incremental view maintenance scheme against naive full re-computations, and to quantify the performance gains. We have been working on an adaptation of the incremental maintenance counting algorithm in the context of SPARQL. Utilizing delta rules, adapted to the algebra of SPARQL, using a combination of join, union, difference, selection and projection rules. Using the difference and join we are able to also implement the OPTIONAL operation in SPARQL. Next to difference, there is also an adaption to the minus operation in SPARQL in our proposed algorithm. We plan to incorporate our techniques in a Web agent operating in the Solid ecosystem. Finally, more research is needed on aggregation, as well as applying incremental view maintenance to aggregators over multiple pods.

References

[1] A. V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, T. Berners-Lee, Solid: a platform for decentralized social applications based on linked data, MIT CSAIL & Qatar Computing Research Institute, Tech. Rep. (2016).

[2] The Solid Team, Solid project, https://solidproject.org/, 2024. Accessed: 2024-04-05.

[3] M. Vandenbrande, M. Jakubowski, P. Bonte, B. Buelens, F. Ongenae, J. Van den Bussche, POD-QUERY: Schema mapping and query rewriting for Solid pods, in: I. Fundulaki, K. Kozaki, et al. (Eds.), Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice, volume 3632 of *CEUR Workshop Proceedings*, 2024.

[4] M. Vandenbrande, Aggregators to realize scalable querying across decentralized data sources, ISWC Doctoral Consortium, 2023.

[5] T. Berners-Lee, Linked data, http://www. w3. org/DesignIssues/LinkedData. html (2007).

[6] W3C Solid Community Group, Solid protocol, https://solidproject.org/TR/protocol, 2022. Accessed: 2024-04-05.

[7] J. A. Blakeley, P.-A. Larson, F. W. Tompa, Efficiently updating materialized views, in:

Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, SIGMOD '86, Association for Computing Machinery, New York, NY, USA, 1986, p. 61–71. URL: https://doi.org/10.1145/16894.16861. doi:10.1145/16894.16861.

[8] X. Qian, G. Wiederhold, Incremental recomputation of active relational expressions, IEEE transactions on knowledge and data engineering 3 (1991) 337–341.

[9] T. Griffin, L. Libkin, H. Trickey, An improved algorithm for the incremental recomputation of active relational expressions, IEEE Transactions on Knowledge and Data Engineering 9 (1997) 508–511.

[10] P.-Å. Larson, J. Zhou, Efficient maintenance of materialized outer-join views, 2007, pp. 56 – 65. doi:10.1109/ICDE.2007.367851.

[11] T. Griffin, B. Kumar, Algebraic change propagation for semijoin and outerjoin queries, SIGMOD Record 27 (1998) 22–27. doi:10.1145/290593.290597.

[12] T. Griffin, L. Libkin, H. Trickey, An improved algorithm for the incremental recomputation of active relational expressions, IEEE Transaction on Knowledge and Data Engineering 9 (1997) 508–511.

[13] C. Koch, et al., DBToaster: Higher-order delta processing for dynamic, frequently fresh views, The VLDB Journal 23 (2014) 253–278.

[14] E. Menendez, M. Casanova, V. Vidal, et al., Incremental maintenance of materialized SPARQL-based linkset views, in: S. Hartmann, H. Ma (Eds.), Proceedings 27th International Conference on Database and Expert Systems Applications, Part II, volume 9828 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 68–83.

[15] A. Gupta, I. S. Mumick, et al., Materialized views: techniques, implementations, and applications, MIT press Cambridge, MA, 1998.