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

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 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

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.

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

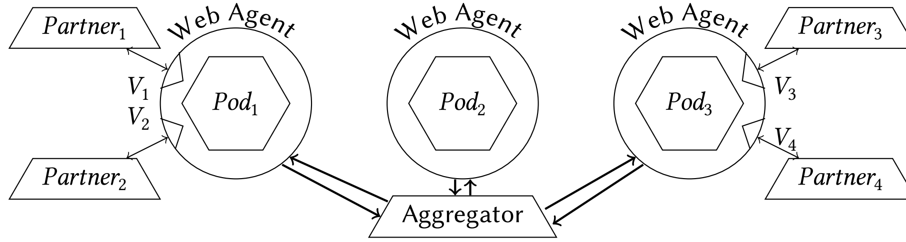

Figure 1: Three pods together with their web agents which handle which trusted partners, such as the aggregator in the illustration, can access which parts of the data. Web Agents provide views V_i which trusted partners can utilize.

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 Descroption Framework), the data standard used in Solid [6].

2. 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.

First, we assess the counting algorithm [7] and its applicability in a SPARQL and linked data setting. It was primarily developed to the maintenance of views containing aggregate functions such as COUNT, SUM, AVG, MIN and MAX. The specific goal of this algorithm is to update 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. 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.

Second, we intend to rely on propagation rules, such as developed in [8] and later improved in [9]. Propagation rules essentially express equivalences between a change in an entire view and the changes within the view, based on the underlying changes in the data. Using such rules, changes in the data can be rewritten efficiently as changes in the view, again avoiding the need for a full re-computation.

We formulate the incremental view maintence 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.

3. 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.

To investigate the algorithms mentioned in the previous section, we express SPARQL queries 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 [7]. We adapt this algorithm to work for RDF data. Now, counts of triples 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.

On the poster, these results are illustrated with worked examples.

4. 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 an 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. Such implementations will be required to run Solid-based applications and ecosystems at web-scale.

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, Podquery: Schema mapping and query rewriting for solid pods, in: ISWC2023, the International Semantic Web Conference, 2023.
[4] M. Vandenbrande, Aggregators to realize scalable querying across decentralized data sources, in: ISWC2023, the International Semantic Web Conference, 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] A. Gupta, I. S. Mumick, et al., Materialized views: techniques, implementations, and applications, MIT press Cambridge, MA, 1998.
[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.