CROISSANT: Centralized Relational Interface for Web-scale SPARQL Endpoints

Takahiro Komamizu, Toshiyuki Amagasa, Hiroyuki Kitagawa
University of Tsukuba
Tsukuba, Japan
taka-coma@acm.org,{amagasa,kitagawa}@cs.tsukuba.ac.jp

ABSTRACT

Searching over Linked Data requires large efforts to users, which include knowing locations of suitable SPARQL endpoints and writing appropriate SPARQL queries in terms of language standards as well as the underlying structure of Linked Data. This situation degrades usability of Linked Data, thus is highly problematic. To resolve this problem, this paper proposes **CROISSANT** which is a centralized view management system for SPARQL endpoints on the Web. CROISSANT stores pre-defined view definitions, and provides a searchable interface for the views to users. To realize CROISSANT, query processing performance is a big issue, because CROISSANT has to communicate with remote SPARQL endpoints and it takes time to receive results. To cope with this issue, this paper proposes four optimization techniques, namely, **view materialization**, **selection push-down**, **projection push-down**, and **view query merge**. Experimental evaluation demonstrates these optimizations improve query processing performance.

CCS CONCEPTS

• **Information systems** → **Data federation tools**; *Semi-structured data*; *Federated databases*; *Browsers*; • **Human-centered computing** → *Accessibility systems and tools*;

KEYWORDS

Linked Data, Relational interface, View-based search, View management, SPARQL endpoints, Query optimization

ACM Reference Format:
Takahiro Komamizu, Toshiyuki Amagasa, Hiroyuki Kitagawa . 2017. CROISSANT: Centralized Relational Interface for Web-scale SPARQL Endpoints. In *iiWAS '17: The 19th International Conference on Information Integration and Web-based Applications & Services, December 4–6, 2017, Salzburg, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3151759.3151803

1 INTRODUCTION

Linked Data (or LD) [2] has pervaded various domains and constitutes a web-scale data network. When users want to search entities, the users require special knowledge for LD such as SPARQL query language [9] as well as vocabularies, locations (or URLs) of SPARQL endpoints, and underlying structures of LD datasets. For example, suppose that a user wants to obtain movie and actor information, there are two steps she must do: (1) she describes her information need in SPARQL, and (2) she posts the SPARQL query onto appropriate SPARQL endpoints which contain movie and actor information (e.g., DBpedia[1], or LinkedMDB[2]). In this situation, she cannot obtain expected information unless she properly knows how to describe her information demand in SPARQL and locations of SPARQL endpoints. Hence, searching over LD is still not an easy tool for non-professional users if they do not understand SPARQL, vocabularies and datasets in LD, and SPARQL endpoints.

To cope with the aforementioned shortage, this paper proposes **CROISSANT**, a Centralized RelatiOnal Interface for Web-Scale SPARQL endpoiNTs. In the proposed model, each view is defined by a view designer who quarries partial information from a SPARQL endpoint, and views are stored in a centralized management server. Views enable users to easily access necessary information without knowledge about SPARQL, vocabularies, and locations of SPARQL endpoints. CROISSANT performs queries on the views and returns results to users.

The naïve approach does not scale because of immature performances of SPARQL endpoints and result transport latencies, thus, this paper introduces three optimization strategies, namely, **view materialization**, **selection push-down**, **projection push-down** and **view query merge**. As discussed on materialized view in relational database [18], materializing views is highly effective to reduce processing costs, when the materialized views are up-to-date. While, SPARQL endpoints are on the Web, thus it is needed to perform remote query processing, which is costly for query processing on SPARQL endpoints as well as result transport. Thus, it is important to reduce the number of results from remote SPARQL endpoints. To realize the reduction, this paper applies selection push-down and projection push-down optimizations. More complicated queries such as joining multiple views takes more transport costs as the number of views in the queries. View query merge optimization copes with this situation, by combining view queries if they are connectable and on the same SPARQL endpoints.

Experimental evaluation ensures CROISSANT is applicable to real-world SPARQL endpoints, and the optimizations achieve better processing time comparing with a naïve approach. As expected from the success of relational database optimizations, materialization optimization works best among the optimizations and a pure execution. Selection push-down also successfully filters unnecessary results on the views and reduces transport costs. View query merge decreases the selectivity of the concatenated queries than

[1]http://dbpedia.org/sparql
[2]http://data.linkedmdb.org/sparql

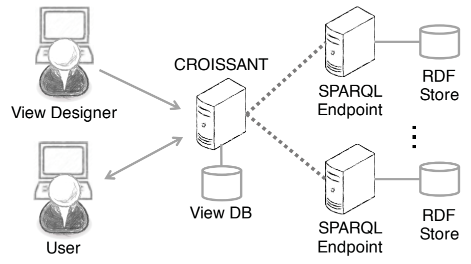

Figure 1: Overview of CROISSANT.

that of original view queries, and succeeds to reduce query execution costs. Additionally, the combination of selection push-down and view query merge optimizations achieves the best performance if users' queries span among several views with filtering conditions.

Contributions of this paper are summarized as follows:

- **CROISSANT**: This paper proposes a centralized relational interface for Web-scale SPARQL endpoints, which manages pre-defined views and realizes user-friendly interfaces.
- **Optimizations**: This paper proposes four optimization techniques, namely, view materialization, selection push-down, projection push-down and view query merge.
- **Experimentation**: CROISSANT and the optimizations are examined on a real-world largest SPARQL endpoint, DBpedia, and shows applicability of CROISSANT as well as effectiveness of the optimization techniques.

2 CROISSANT: PROPOSED METHOD

CROISSANT is a centralized view management system for SPARQL endpoints, which provides relational interfaces based on pre-defined views on SPARQL endpoints. Figure 1 gives an overview of CROISSANT. CROISSANT involves two actors, **view designers** and **users**. The former define views on SPARQL endpoints and store views on CROISSANT, and the latter ask queries (i.e., SQL queries) on views to CROISSANT and receive results of the queries. The pre-defined views are managed by CROISSANT which keeps the views in a view database. CROISSANT accepts user queries written in SQL, rewrites them into SPARQL queries, processes the queries onto corresponding SPARQL endpoints, and returns the results of the queries to users.

View designers describe views by SPARQL, where view composes a quadruplet defined in Definition 2.1. Though schemas of views can be any format like relational, XML, JSON, and so on, this paper assumes the format is relational because relational format is one of the most common formats for Web applications and relational format directly fits to returned results from SPARQL queries.

Definition 2.1 (View Definition). A view definition is composed of the following quadruplet.

$$\langle name, schema, endpoint\_url, SPARQL\_query \rangle$$

where *name* is the identifier of the view, *schema* is a relational schema of the view, *endpoint_url* is a URL of the SPARQL endpoint with which the view is associated, and *SPARQL_query* is the **SELECT** query which quarries a part of RDF data in the endpoint. □

Based on the view definitions discussed above, CROISSANT handles the view definitions as well as queries from users. To realize this, CROISSANT consists of the following components: (1) view design support for view designers, (2) SQL interface for users, (3) query executor, (4) query optimization, and (5) view management. *View design support* component aims at supporting view designers building view definitions, *SQL interface* component allows users to post SQL queries onto CROISSANT, and *query executor* component first translates the given SQL queries into SPARQL, then performs the SPARQL queries onto corresponding SPARQL endpoints, and answers the SQL queries based on the results from SPARQL endpoints. In addition, *query optimization* component improves the performance of query processing based on heuristic approaches (shown in Section 3), and *view management* component allows view designers to edit view definitions like addition, removals, updates of view definitions and checks syntaxes and rules of view definitions. Note that this paper mainly deals with the query optimization, therefore, the following sections introduces the proposal of query optimizations.

3 VIEW-BASED QUERY OPTIMIZATION

In a naïve method, user queries are performed as the following three steps: CROISSANT (1) extracts relevant views for the input user query, (2) performs the view queries to obtain individual results, and (3) processes user queries over the results. The first step determines SPARQL endpoints and corresponding SPARQL queries related with user queries. Then, CROISSANT executes the SPARQL queries on corresponding SPARQL endpoints to materialize contents of the views onto the local storage. Finally, for the materialized views, CROISSANT executes the user queries to obtain results. The naïve method has critical performance drawbacks as follows:

- **Execution cost**: The naïve executes view query on SPARQL endpoints. The cost depends on the query processing performances of SPARQL endpoints.
- **Transport cost**: Results of view queries must be transported to the local server. The cost depends on the size of the results and the network transportation latency.

In order to overcome the drawbacks of the naïve method, CROISSANT employs four optimization strategies for performing user queries. The individual optimization strategies will be discussed in the next sections.

3.1 View Materialization

Performing SPARQL queries onto remote SPARQL endpoints is costly comparing with SQL performing on local server (i.e., CROISSANT server), and, in such situation, materialized views realize significant performance improvements. If the latest data of a view are available on the local server, CROISSANT can perform SQL query processing directly to materialized views without accessing to remote SPARQL endpoints. CROISSANT in this paper assumes relational views on SPARQL endpoints, thus each view can be materialized into local relational database. For other formats, view data are stored in a corresponding databases, for example, MongoDB[3] for JSON views, and BaseX[4] for XML views. Materialized

[3]https://www.mongodb.com/
[4]http://basex.org/

views suffer from view update problem and query processing using materialized and non-materialized views.

In a traditional database situation where databases are assumed to be local, trigger functionalities help notify changes to database management systems and run program to restore the query results to materialized view. This is obviously not applicable to SPARQL endpoints on the Web, because they are not under control of CROISSANT. Some researches have attempted to realize the notification functionality (like [17, 22]), it is possible to use the functionality to know when to update the materialized views, however this is still an open problem.

Checking update on SPARQL endpoints is close to checking updates on Web pages in Web crawling applications. In Web crawling researches, Poisson distribution is a popular distribution of page updates [5, 6]. Web crawlers can check cached Web pages based on the Poisson distribution. Observations of LD dynamics [12, 19] can be a help to introduce when to check updates of LD datasets, however, these observations are still not much helpful to characterize LD datasets in terms of update frequency. Consequently, periodical checking for updates of LD datasets is a compromising approach. If the update frequency on an LD data source follows a probability distribution (like Poisson distribution), Web crawling strategies are applicable, that is, checking updates of view query results in periods derived from the probability distribution.

Materializing views have a trade-off w.r.t. database update frequency. If the frequency is quite high (more than query response, for example), it is not good option to materialize because of freshness of data in views. If a view is not materialized, query processing on views suffer from time of view query processing and transport of results. Thus, if the update frequency of original data of a view in SPARQL endpoints is high, the view is kept non-materialized and CROISSANT executes view queries whenever users execute their queries. While, if the update frequency is low, the view is materialized eagerly and CROISSANT executes user queries directly on the materialized relational view.

3.2 Selection Push-down

As optimization techniques in relational databases [18], in which selection push-down is a typical and significant approach to reduce the number of relevant results on query operators, CROISSANT employs selection push-down approach. The number of records which meet selection conditions is typically significantly smaller than that of records in the tables. This observation indicates that selection conditions should performed as soon in query plans as possible, especially when join operations are included in the query.

CROISSANT involves applying selection push-down from user queries to view queries. As user queries are written in SQL language, selection conditions on user queries can be represented as selection operators. CROISSANT then classifies the extracted selection operations into views (obviously, once CROISSANT knows chosen relations and selection conditions on attributes of them, it also knows corresponding views and SPARQL variables on the views). Finally, CROISSANT puts FILTER clauses into view queries.

Pushed-down selection conditions should include θ binary comparators (i.e., $<$, $\leq$, $=$, $\geq$, and $>$) to compare an attribute with numeric values and regular expression-based comparators for textual values. In order to deal with these operators, CROISSANT includes rules to convert selection conditions into FILTER expressions. This paper defines three basic rules in Table 1. The first rule is about numeric conditions which directly put in the FILTER clause. The second rule is about textual equality conditions. Because textual values in RDF typically have language information (e.g., `@en` means English text), direct comparisons do not work, `str(·)` function is applied to return language-ignored textual values of variables. The third rule is about regular expression-based textual matching. Fortunately, SPARQL includes regular expression function called `regex(·)` which judges whether textual values match with input regular expressions.

Table 1: Conversion from selection conditions to SPARQL FILTER clauses.

Comparator	Variable type	FILTER expression
θ	Numeric	`FILTER (`*`variable θ value`*`)`
=	Textual	`FILTER (str(`*`variable`*`) = `*`value`*`)`
like	Textual	`FILTER regex(`*`variable, value`*`)`

3.3 Projection Push-down

Projection push-down is also a traditional query optimization technique in relational database researches, which puts projection conditions onto sub-queries in order to reduce the size of tuples to be returned. The intermediate projection conditions which do not affect processing of intermediate results (such as join results) and final results (filtered out attributes) can be removed from the sub-queries. In such cases, projection conditions on superior queries can be pushed down into the sub-queries.

The same idea can be applied to CROISSANT situation, that is, CROISSANT pushes the projection conditions on users' queries into the view SPARQL queries and remove other variables if the conditions do not affect on both intermediate and final results. Basically, CROISSANT pushes down projection conditions into corresponding view queries. When CROISSANT attempts to remove projection variables in view queries, it cares if the variables are included in join conditions in users' SQL queries. If included, CROISSANT remains it on the view queries, and remove otherwise.

3.4 View Query Merge

Query performance of user queries depends on the numbers of views included in the user queries because the numbers of views are corresponding with that of SPARQL queries to be performed. However, some of views may be of same SPARQL endpoints. If the graph patterns in SPARQL queries in the view queries can be connected (i.e., share same variables), these queries can be integrated as a single SPARQL query. As a result of combination, there are two benefits: (1) as graph patterns get complicated, the number of results matching with the query can be decreased, in consequence, result transport cost can be reduced; and (2) (relational) join processing on the local server (typically, this is costly) can be ignored, because the single SPARQL query returns the joined results.

Given a view set V of a user query Q, CROISSANT first extracts join conditions between views, then merges corresponding views with the join conditions. CROISSANT accepts a common join operation called θ-join which conditions can be either $<$, $\leq$, $=$, $\geq$, or $>$. CROISSANT merges views on the same SPARQL endpoints with the join conditions as following steps: (1) CROISSANT extracts returned variables for each view, makes a list of the variables for merged views with duplicate elimination, and puts the list into SELECT clause; (2) CROISSANT extracts graph patterns in WHERE clause for each view, eliminates duplicates of the graph patterns, and puts them into WHERE clause; (3) Finally, CROISSANT adds FILTER clauses for all join conditions.

4 EXPERIMENTAL EVALUATION

The experimental evaluation attempts to answer the following questions.

- **Q1:** *Is CROISSANT applicable to real-world SPARQL endpoints in terms of usability?*
- **Q2:** *How much performance improvements can be achieved when the proposed optimization techniques are applied?*

In order to answer these questions, CROISSANT runs on a local server holding a relational database (PostgreSQL[5] in this implementation), this paper designs views and queries on real-world SPARQL endpoints on the Web, and evaluates CROISSANT in terms of processing performance. The following sections explain experimental settings (Section 4.1) and evaluation results (Section 4.2).

4.1 Experimental Settings

In order to answer **Q1**, CROISSANT is applied to the most popular SPARQL endpoint, DBpedia which is the largest knowledge base on the Web and contains large variations of entities. SPARQL endpoint of DBpedia is very stable and reasonably fast, thus it is good for evaluation.

In order to answer **Q2**, are designed to observe whether optimizations work. The queries include two simple selection queries (`q0` and `q1`) and two join queries with and without selection conditions (`q2` and `q3`). The former two queries are used for observing effectiveness of selection push-down, and the latter two queries are used for observing effectiveness of view query merge as well as selection push-down.

For each query, this experiment prepares five optimization strategies and measures the average time for executions and transports over 10 trials for each optimization strategy. The optimization strategies are (**P**) pure execution, (**M**) materialization, (**SP**) selection push-down, (**MG**) view query merge, and (**MGSP**) view query merge with selection push-down.

- (**P**): CROISSANT purely executes view queries of input queries, stores the results into the local storage, and executes user queries over the local storage.
- (**M**): CROISSANT materializes all views into the local storage in advance, and, then executes on demand user queries on the materialized views.
- (**SP**): CROISSANT performs selection push-down if available, executes view queries with pushed down selection conditions, stores the results and executes user queries over the local storage.

[5]https://www.postgresql.org/

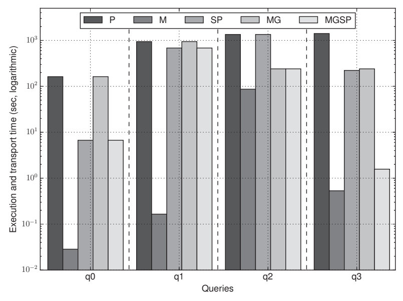

Figure 2: Performance comparison.

- (**MG**): CROISSANT performs view query merge if possible, executes merged queries, stores the results into the local storage, and executes user queries.
- (**MGSP**): This is concatenation of (**SP**) and (**MG**).

4.2 Results

Figure 2 shows processing times for each query in terms of five optimization strategies, and the figure indicates that CROISSANT is applicable to real-world SPARQL endpoints and optimization improves query processing performance. As expected, materialization works the best among optimizations for all queries in this experiment. In the case where materialization is not applicable, selection push-down and view query merge are good optimization choices as shown for `q0`, `q2`, and `q3`. In particular, the combination of selection push-down and view query merge (**MGSP**) for `q3` achieves significant improvements from individual optimizations (i.e., **MG** and **SP**).

Consequently, answers for the aforementioned questions are:

- **A1**: *CROISSANT is applicable to a representative real-world SPARQL endpoint. In terms of query performance, CROISSANT is reasonably applicable if materialization is doable, but CROISSANT requires more improvements in some cases like queries with high selectivity.*
- **A2**: *Optimizations work well, materialization is quite effective if available, selection push-down and view query merge are also effective, and their combination is more. In the case where materialization is not doable, CROISSANT does not work well for high selectivity queries (like `q1`).*

5 RELATED WORK

Views on RDF have been studied in these decades [7, 20, 21]. For example, Schenk et al. [20] have proposed *Networked Graphs* which is a means of describing RDF graphs, it acts as views on RDF data, Shaw et al. [21] have defined a view definition language called *vSPARQL*, and Etcheverry et al. have surveyed views over RDF datasets [7]. RDF views have been used in various purposes: Abel et al. [1] utilize views for access control on RDF datasets, many works have attempted to improve query performances by view materialization.

View materialization for RDF datasets have been studied for improving complex queries, such as aggregation queries [10, 11] and join queries [4], as well as for view selection [8, 11, 14, 15]. Hung et al. [10] have proposed views for aggregation over RDF data and how to maintain the aggregated values on the views. The basic model of them is local RDF database, and their approach is not easily applicable to SPARQL endpoints on the Web. Ibragimov et al. [11] have proposed an optimization mechanism for aggregate SPARQL queries using materialized views. They assume that analytical processing over RDF data, thus they select views to be materialized based on the data cube lattice. As well, in their view selection algorithm, they select the best view in order to perform user queries, meaning that they do not taking multiple view optimization into their consideration. Castillo et al. [4] have discussed materialized views over RDF databases in order to process RDF data efficiently by reducing join processing. They assume local RDF database, and their work is not directly applicable to SPARQL endpoints on the Web. Goasdoué et al. [8] have proposed view selection algorithm for RDF databases based on a given set of SPARQL queries and query workloads in order to optimize the query cost, view storage, and view maintenance cost. Le et al. [14] have proposed an SPARQL query rewriting scheme over SPARQL views, which automatically choose a set of views matching with the input SPARQL queries and rewrite the queries using the view queries in order to obtain results from RDF datasets under the views. Lynden et al. [15] have proposed a hybrid approach to LD query processing, which simultaneously querying over Web of LD and SPARQL endpoints to obtain fresh information. Even though they do not use views for SPARQL endpoints, their idea can be combined to the query processing mechanism which may query both materialized and virtual views simultaneously.

The previous work [13] have proposed JSON views over SPARQL endpoints. For the JSON views, query processing system accepts Language-Integrated Query (or LINQ)[6]. The query processing system executes view queries on SPARQL endpoints and then perform LINQ queries over the views. The previous work [13] was a prototypical framework and did not include any optimization methodologies in order to improve query processing performance. While, this paper arranges target query language from LINQ to SQL because SQL is more popular in Web application development, and proposes optimization strategies over query processing.

Maintaining materialized views is an important problem, however, dynamics of SPARQL endpoints (i.e., how frequently SPARQL endpoints are updated) are still not evident. There are few works dealing with update detection for SPARQL endpoints [17, 22]. Passant and Mendes [17] have proposed push notification mechanism on SPARQL endpoints with PubSubHubbub protocol[7]. Teymourian et al. [22] have proposed an optimization schema for subscribing DBpedia data through DBpedia live[8]. If other SPARQL endpoints are able to provide change event streams like DBpedia live, their approach may be available, however, most of SPARQL endpoints are not not available on that functionality for now.

Other kind of views called *linkset views* [3, 16] are also interesting view-related research. Linkset views are defined as relationship views among entities which are not explicitly connected. Menendez et al. [16] have studied on how to update the linkset views with time evolution. To include linkset views is left to the future work because maintaining linkset views needs more effort.

[6]https://msdn.microsoft.com/en-us/en-us/library/bb308959.aspx
[7]http://code.google.com/p/pubsubhubbub/
[8]http://live.dbpedia.org/

6 CONCLUSION

This paper proposes CROISSANT, an ostensible (relational) interface for SPARQL endpoints using views. To cope with performance issues on CROISSANT, this paper proposes four optimization techniques, namely, view materialization, selection push-down, projection push-down and view query merge. The experiments show that the view materialization realizes the quickest processing if materialization is available. Otherwise, the selection push-down performs better as the selectivity of queries decreases, and the view query merge improves query performance when queries are complicated (joining three or more tables on the same SPARQL endpoints).

Acknowledgement

This research was partly supported by the program *Research and Development on Real World Big Data Integration and Analysis* of RIKEN, Japan.

REFERENCES

[1] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. 2007. Enabling Advanced and Context-Dependent Access Control in RDF Stores. In *ISWC'07 + ASWC'07*. 1–14.
[2] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.* 5, 3 (2009), 1–22.
[3] Marco A. Casanova, Vânia Maria Ponte Vidal, Giseli Rabello Lopes, Luiz André P. Paes Leme, and Lívia Ruback. 2014. On Materialized sameAs Linksets. In *DEXA'14*. 377–384.
[4] Roger Castillo, Christian Rothe, and Ulf Leser. 2010. RDFMatView: Idexing RDF Data for SPARQL Queries. In *Technical Report 234, Humboldt Universität zu Berlin*.
[5] Junghoo Cho and Hector Garcia-Molina. 2000. Synchronizing a Database to Improve Freshness. In *SIGMOD'00*. 117–128.
[6] Edward G Coffman, Zhen Liu, and Richard R Weber. 1997. *Optimal Robot Scheduling for Web Search Engines.* Ph.D. Dissertation. INRIA.
[7] Lorena Etcheverry and Alejandro A. Vaisman. 2012. Views over RDF Datasets: A State-of-the-Art and Open Challenges. *CoRR* abs/1211.0224 (2012).
[8] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. 2011. View Selection in Semantic Web Databases. *PVLDB* 5, 2 (2011), 97–108.
[9] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. (2013). https://www.w3.org/TR/sparql11-query/.
[10] Edward Hung, Yu Deng, and V. S. Subrahmanian. 2005. RDF Aggregate Queries and Views. In *ICDE'05*. 717–728.
[11] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2016. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *ISWC'16*. 341–359.
[12] Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O'Byrne, and Aidan Hogan. 2013. Observing Linked Data Dynamics. In *ESWC'13*. 213–227.
[13] Kazumasa Kumamoto, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2015. A System for Querying RDF Data Using LINQ. In *NBiS'15*. 452–457.
[14] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. 2011. Rewriting Queries on SPARQL Views. In *WWW'11*. 655–664.
[15] Steven J. Lynden, Isao Kojima, Akiyoshi Matono, Akihito Nakamura, and Makoto Yui. 2013. A Hybrid Approach to Linked Data Query Processing with Time Constraints. In *LDOW'13*.
[16] Elisa Menendez, Marco A. Casanova, Vânia Maria P. Vidal, Bernardo Pereira Nunes, Giseli Rabello Lopes, and Luiz André P. Paes Leme. 2016. Incremental Maintenance of Materialized SPARQL-Based Linkset Views. In *DEXA'16*. 68–83.
[17] Alexandre Passant and Pablo N. Mendes. 2010. sparqlPuSH: Proactive Notification of Data Updates in RDF Stores Using PubSubHubbub. In *SFSW'10*.
[18] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems*. McGraw-Hill.
[19] Yannis Roussakis, Ioannis Chrysakis, Kostas Stefanidis, Giorgos Flouris, and Yannis Stavrakas. 2015. A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets. In *ISWC'15*. 495–512.
[20] Simon Schenk and Steffen Staab. 2008. Networked Graphs: A Declarative Mechanism for SPARQL Rules, SPARQL Views and RDF Data Integration on the Web. In *WWW'08*. 585–594.
[21] Marianne Shaw, Landon Fridman Detwiler, Natalya Fridman Noy, James F. Brinkley, and Dan Suciu. 2011. vSPARQL: A view definition language for the semantic web. *Journal of Biomedical Informatics* 44, 1 (2011), 102–117.
[22] Kia Teymourian, Alexandru Todor, Wojciech Lukasiewicz, and Adrian Paschke. 2015. Optimized Processing of Subscriptions to DBpedia Live. In *BIS'15*. 293–307.