Scheduling Refresh Queries for Keeping Results from a SPARQL Endpoint Up-to-Date
(Short Paper)

Magnus Knuth[1], Olaf Hartig[2], and Harald Sack[1]

[1] Hasso Plattner Institute, University of Potsdam, Germany
{magnus.knuth|harald.sack}@hpi.de

[2] Dept. of Computer and Information Science (IDA), Linköping University, Sweden
olaf.hartig@liu.se

Abstract. Many datasets change over time. As a consequence, long-running applications that cache and repeatedly use query results obtained from a SPARQL endpoint may resubmit the queries regularly to ensure up-to-dateness of the results. While this approach may be feasible if the number of such regular refresh queries is manageable, with an increasing number of applications adopting this approach, the SPARQL endpoint may become overloaded with such refresh queries. A more scalable approach would be to use a middle-ware component at which the applications register their queries and get notified with updated query results once the results have changed. Then, this middle-ware can schedule the repeated execution of the refresh queries without overloading the endpoint. In this paper, we study the problem of scheduling refresh queries for a large number of registered queries by assuming an overload-avoiding upper bound on the length of a regular time slot available for testing refresh queries. We investigate a variety of scheduling strategies and compare them experimentally in terms of time slots needed before they recognize changes and number of changes that they miss.

1 Introduction

Many datasets on the Web of Data reflect data related to current events or ongoing activities. Thus, such datasets are dynamic and evolve over time [9]. As a consequence, query results that have been obtained from a SPARQL endpoint may become outdated. Therefore, long-running applications that cache and repeatedly use query results have to resubmit the queries regularly to ensure up-to-dateness of the results.

There would be no need for such regular tests if SPARQL endpoints would provide information about dataset modifications. There exist manifold approaches for providing such information. Examples are cache validators for SPARQL requests (using HTTP header fields such as `Last-Modified` or `ETag`) [15] and publicly available dataset update logs (as provided by DBpedia Live at http://live.dbpedia.org/changesets/). Unfortunately, existing SPARQL endpoints rarely support such approaches [6], nor is

This work was funded by grants from the German Government, Federal Ministry of Education and Research for the project D-Werft (03WKCJ4D).

update information provided in any other form by the dataset providers. The information needed has to be generated by the datastore underlying the SPARQL endpoint or by dataset wrappers that exclusively control all the updates applied to the dataset, which is often not possible, e.g. in the case of popular RDB2RDF servers, as they typically work as one-way RDF exporters.

With an increasing number of applications re-executing their queries, the SPARQL endpoint might become overloaded with the refresh queries. A more scalable approach would be to use a middle-ware component at which the applications register their queries and get notified updates once the query results have changed. A main use case of such a middle-ware is the sparqlPuSH approach to provide a notification service for data updates in RDF stores [11]. sparqlPuSH tracks changes of result sets that are published as an RSS feed and broadcasted via the PubSubHubbub protocol. However, the existing implementation of sparqlPuSH is limited to the particular use case of microposts and circumvents the problem of detecting changes by expecting dataset updates to be performed via the sparqlPuSH interface. To generalize the idea of sparqlPuSH scheduling the re-evaluation of SPARQL queries has been identified as an unsolved research problem [8].

In this paper, we study this problem of scheduling refresh queries for a large number of registered SPARQL queries; as an overload-avoiding constraint we assume an upper bound on the length of time slots during which sequences of refresh queries can be run. We investigate various scheduling strategies and compare them experimentally. For our experiments, we use a highly dynamic real-world dataset over a period of three months, in combination with a dedicated set of queries. The dataset (DBpedia Live) comprises all real-time changes in the Wikipedia that are relevant for DBpedia.

The main contributions of the paper are an empirical evaluation of a corpus of real-world SPARQL queries regarding result set changes on a dynamic dataset and an experimental evaluation of different query re-evaluation strategies. Our experiments show that the change history of query results is the main influential factor, and scheduling strategies based on the extent of previously recognized changes (dynamics) and an adaptively allocated maximum lifetime for individual query results provide the best performances.

The paper is structured as follows: Sec. 2 discusses related work. Sec. 3 provides definitions and prerequisites. These are needed for Sec. 4 which introduces the scheduling strategies used for the experiments. Sec. 5 and Sec. 6 describe the experimental setup and the applied evaluation metrics, respectively. In Sec. 7 we present the experimental results and discuss them in Sec. 8.

2 Related Work

A variety of existing applications is related to change detection of query results on dynamic RDF datasets, such as (external) query caching [10], partial dataset update [2], as well as notification services [11]. However, even though Williams and Weaver show how the `Last-Modified` date can be computed with reasonable modifications to a state-of-the-art SPARQL processor [15], working implementations are rare. In fact, Kjernsmo has shown in an empirical survey that only a miniscule fraction of public SPARQL endpoints actually support caching mechanisms on a per-query basis [6].

To overcome this lack of direct cache indicators, alternative approaches are required to recognize dataset updates. The most common approach is to redirect updates through a wrapper that records all changes [10,11]. However, this approach is not applicable for datasets published by someone else. If data publishers provide information on dataset updates, this information can be analyzed. For instance, Endris et al. introduce an approach to monitor the changesets of DBpedia Live for relevant updates [2] (such a changeset is a log of removed and inserted triples). Tools for dataset update notification, such as *DSNotify* [12] and *Semantic Pingback* [14], are available but rarely deployed.

Since the aforementioned cache indicators and hints for change detection are missing almost entirely in practice, we rely on re-execution of queries. Apparently, such an approach causes overhead in terms of additional network traffic and server load.In order to reduce this overhead we investigate effective scheduling strategies in this paper. A similar investigation in the context of updates of Linked Data has been presented by Dividino et al. [1]. The authors show that change-aware strategies are suitable to keep local *data caches* up-to-date. We also evaluate a strategy adopted from Dividino et al.'s *dynamicity* measure. We observe that, in our context, this strategy performs well for highly dynamic queries, but it is prone to starvation for less dynamic queries.

Query result caches are also used for database systems where the main use case is to enhance the scalability of backend databases for dynamic database-driven websites. The most prominent system is *Memcached*[3] which supports the definition of an expiration time for individual cache entries, as well as local cache invalidation, e. g. when a client itself performs an update. Consequently, updates from other sources cannot be invalidated. More sophisticated systems, such as the proxy-based query result cache *Ferdinand* [3], use update notifications to invalidate local caches.

3 Preliminaries

In this paper we consider a dynamic dataset, denoted by $\mathcal{D}$, that gets updated continuously or in regular time intervals. We assume a sequence $\vec{\mathcal{T}} = (t_1, t_2, \ldots, t_n)$ of consecutive points in time at which the dataset constitutes differing revisions. Additionally, we consider a finite set Q of SPARQL queries. Then, for every time point t_i in $\vec{\mathcal{T}}$ and for every query $q \in Q$, we write $\mathsf{result}(q, i)$ to denote the query result that one would obtain when executing q over $\mathcal{D}$ at t_i. Furthermore, let $C_i \subseteq Q$ be the subset of the queries whose result at t_i differs from the result at the previous time point t_{i-1}, i.e.,

$$C_i = \big\{q \in Q \mid \mathsf{result}(q, i) \neq \mathsf{result}(q, i-1)\big\}.$$

The overall aim is to identify a greatest possible subset of C_i at each time point t_i. A trivial solution to achieve this goal would be to execute all queries from Q at all time points. While this exhaustive approach may be possible for a small set of queries, we assume that the size of Q is large enough for the exhaustive approach to seriously stress, or even overload, the query processing service. Therefore, we consider an additional politeness constraint that any possible approach has to satisfy. For the sake of simplicity, in this paper we use as such a constraint an upper bound on the size of the time

[3] http://www.memcached.org/

slots within which approaches are allowed to execute a selected sequence of queries for the different time points. Hereafter, let $K_{\mathsf{maxExecTime}}$ be this upper bound, and, for any possible approach, let $E_i \subseteq Q$ be the (refresh) queries that the approach executes in the time slot for time point t_i. Hence, if we let $\mathsf{execTime}(q, i)$ denote the time for executing q over the snapshot of $\mathcal{D}$ at t_i, then for all past time points we have

$$K_{\mathsf{maxExecTime}} \geq \sum_{q \in E_i} \mathsf{execTime}(q, i).$$

To select a sequence of queries to be executed within the time slot for a next time point, the approaches may use any kind of information obtained by the query executions performed during previous time slots for earlier time points. For instance, to select the sequence of queries for a time point t_i, an approach may use any query result $\mathsf{result}(q, j)$ with $j < i$ and $q \in E_j$, but it cannot use any $\mathsf{result}(q', j')$ with $q' \notin E_{j'}$ or with $j' \geq i$.

As a last preliminary, in the definition of some of the approaches that we are going to introduce in the next section we write $\mathsf{prevExecs}(q, i)$ to denote the set of all time points for which the corresponding approach executed query $q \in Q$ before arriving at time point t_i; i.e. $\mathsf{prevExecs}(q, i) = \{j < i \mid q \in E_j\}$. In addition, we write $\mathsf{lastExec}(q, i)$ to denote the most recent of these time points, i.e. $\mathsf{lastExec}(q, i) = \max\big(\mathsf{prevExecs}(q, i)\big)$.

4 Scheduling Strategies

This section presents the scheduling strategies implemented for our evaluation. We begin by introducing features that may affect the behavior of such strategies.

Typically, dataset providers do not offer any mechanism to inform clients about data updates, neither whether the data has changed nor to what extent. Therefore, we focus on scheduling strategies that are dataset agnostic, i. e. strategies that do not assume information about what has changed since the last query execution. Hence, all features that such a strategy can exploit to schedule queries for the next refresh time slot originate from (a) the queries themselves, (b) an initial execution of each query, and (c) the ever growing history of successful executions of the queries during previous time slots.

We have implemented seven scheduling policies known from the literature. We classify them into two groups: *non-selective* and *selective* policies. By using a *non-selective* scheduling policy, potentially all registered queries are evaluated according to a ranking order until the execution time limit ($K_{\mathsf{maxExecTime}}$) has been reached. For every time point t_i in $\overrightarrow{\mathcal{T}}$, a new ranking for all queries is determined. The queries are ranked in ascending order using a ranking function $\mathsf{rank}(q, i)$. In a tie situation, the decision is made based on the age of the query, and finally the query id.

Round-Robin (RR) treats all queries equal disregarding their change behavior and execution times. It executes the queries for which the least current result is available.

$$\mathsf{rank}_{RR}(q, i) = \frac{1}{i - \mathsf{lastExec}(q, i)} \tag{1}$$

Shortest-Job-First (SJF) prefers queries with a short estimated runtime (to execute as many queries per time slot as possible). The runtime is estimated using the median

value of runtimes from previous executions. Additionally, the exponential decay function $e^{-\lambda(i-\mathsf{lastExec}(q,i))}$ is used as an aging factor to prevent starvation.

$$\mathsf{rank}_{SJF}(q,i) = e^{-\lambda(i-\mathsf{lastExec}(q,i))}\mathrm{median}_{j\in\mathsf{prevExecs}(q,i)}\big(\mathsf{execTime}(q,j)\big) \quad (2)$$

Longest-Job-First (LJF) uses the same runtime estimation and aging as SJF but prefers long estimated runtimes, assuming such queries are more likely to produce a result.

$$\mathsf{rank}_{LJF}(q,i) = \frac{e^{-\lambda(i-\mathsf{lastExec}(q,i))}}{\mathrm{median}_{j\in\mathsf{prevExecs}(q,i)}\big(\mathsf{execTime}(q,j)\big)} \quad (3)$$

Change-Rate (CR) prioritizes queries that have changed most frequently in the past. A decay function $e^{-\lambda t}$ is used to weight the change added by its respective age.

$$\mathsf{rank}_{CR}(q,i) = \sum_{j\in\mathsf{prevExecs}(q,i)} \left(e^{-\lambda(i-j)} * \mathsf{change}(q,i)\right), \quad (4)$$

$$\text{where:}\quad \mathsf{change}(q,i) = \begin{cases} 1 & \text{if } \mathsf{result}(q,j) \neq \mathsf{result}(q,\mathsf{lastExec}(q,j)), \\ -1 & \text{else.} \end{cases} \quad (5)$$

Dynamics-Jaccard (DJ) has been proposed as a best-effort scheduling policy for dataset updates [1]. Here, for `DESCRIBE` and `CONSTRUCT` queries we compute the *Jaccard distance* on RDF triples, and on the query solutions for `SELECT` queries. For `ASK` queries, the distance is either 0 or 1.

$$\mathsf{rank}_{DJ}(q,i) = \sum_{j\in\mathsf{prevExecs}(q,i)} \left(e^{-(i-j)} * \mathsf{jaccard}(q,j)\right) \quad (6)$$

$$\text{where:}\quad \mathsf{jaccard}(q,j) = 1 - \frac{\left|\mathsf{result}(q,j) \cap \mathsf{result}(q,\mathsf{lastExec}(q,j))\right|}{\left|\mathsf{result}(q,j) \cup \mathsf{result}(q,\mathsf{lastExec}(q,j))\right|} \quad (7)$$

Instead of ranking all queries, the *selective* scheduling policies select a (potentially ranked) subset of queries for evaluation at a given point in time t_i. Queries from this subset that do not get evaluated due to the execution time limit ($K_{\mathsf{maxExecTime}}$) are privileged in the next time slot t_{i+1}.

Clairvoyant (CV) is assumed to have full knowledge of all query results at every point in time and, thus, is able to determine the optimal schedule.

Time-To-Live (TTL) determines specific time points when a query should be executed. To this end, each query is associated with a value indicating a time interval after which the query needs to be re-evaluated. After an evaluation, if the query result has changed, this time-to-live value is divided in half or, alternatively, reset to the initial value of 1; if the result did not change, the value is doubled up to a fixed maximum value (max). We investigate different values as maximum time-to-live.

5 Experimental Setup

This section describes the test setup with which we evaluated the effectiveness of the scheduling strategies, and in the next section we introduce the evaluation metrics.

For our experiments we use the *DBpedia Live* dataset [5] because it provides continuous fine-grained changesets, which are necessary to reproduce a sufficient number of dataset revisions. Moreover, while *DBpedia Live* and *DBpedia* share the same structural backbone (both make use of the same vocabularies and are extracted from English Wikipedia articles), the main difference is that the real-time extraction of *DBpedia Live* makes use of different article revisions. Therefore, queries for *DBpedia* work alike for *DBpedia Live*. We selected the three-months period August–October 2015 for replaying the changesets, starting from a dump of June 2015 (http://live.dbpedia.org/dumps/dbpedia_2015_06_02.nt.gz) applied with subsequent updates for June and July 2015. In total we have 2,208 hourly updates for our three-months period (92 days * 24 hours), and there are 437 revisions (hours) without any changes. The dataset size varies between 398M and 404M triples for the different revisions.

To perform SPARQL query executions on a dynamic dataset it is essential to use queries that match the dataset. We use a set of real-world queries from the *Linked SPARQL Queries dataset* (LSQ) [13] which contains 782,364 queries for DBpedia. We randomly selected 10,000 queries from LSQ after filtering out those having a runtime of more than 10 minutes or producing parse or runtime errors. The query set contains 11 `DESCRIBE`, 93 `CONSTRUCT`, 438 `ASK`, and 9458 `SELECT` queries, and is available at https://semanticmultimedia.github.io/RefreshQueries/data/queries.txt. DBpedia Live changes gradually, but obviously the structural backbone of DBpedia remains. As a result, 4,423 out of our 10,000 queries deliver a non-empty query result on the first examined revision (4,440 over all examined revisions). Concerning *result changes*[4] we observe that only a small fraction of the queries is affected by the dataset updates (up to 32 queries per revision, 352 queries within all revisions). Furthermore, we observe that query results may also change after being constant for a long time. The majority (191) of the 352 queries affected by the dataset updates change exactly once, 38 queries change twice, and we recognize that the query results change in very irregular intervals with a high variation between the individual queries. The overall runtime of all queries per revision ranges from 440 to 870 seconds, whereas the runtime for all affected queries ranges up to 50.1 seconds. For more details about the characteristics of the dataset revisions and the queries we refer to the extended version of this paper [7].

The dataset replay and the query executions have been performed on a 48-core Intel(R) Xeon(R) CPU E5-2695 v2 @2.40GHz and an OpenLink Virtuoso Server 07.10 with 32GB reserved RAM. We provide the data gathered from the experiments in form of a MySQL database dump and an RDF dump with the query executions as planned by the evaluated strategies[5].

6 Evaluation metrics

An ideal scheduling strategy should satisfy a number of requirements:

[4] We consider the result of a query as *changed* if it is not isomorphic to the result returned for the query in the previous evaluation. For queries that use the `ORDER BY` feature we also check for an equal bindings sequence. If `ORDER BY` is not used in the query, the binding order is ignored as SPARQL result sets are then expected in no specific order [4].

[5] Both datasets are available at https://semanticmultimedia.github.io/RefreshQueries/

– *Effectiveness*: It should only evaluate queries that have changed, which reduces unnecessary load to the SPARQL endpoint.
– *Efficiency*: It should evaluate queries that have changed as soon as possible, which reduces the out-of-date time and helps to not miss result changes.
– *Avoid starvation*: Results of queries that are susceptible to change (i.e., there is no reason to believe the query will always produce the same result) may change at any point in time even if the results have been constant so far. It should be ensured that such queries are executed at some point.

To compare the query execution strategies we simulate their query selection with different configurations over all 2,208 dataset revisions ($t_1, \ldots, t_{2208}$). The initial query results $\{\forall q \in Q : \mathsf{result}(q, 0)\}$ for $t_0 < 08/01$ are available to every scheduling strategy right from the start. We compute the following key metrics:

Total query executions number of query executions performed.
Irrelevant executions query executions without recognizing a change, equals to the total number of executions minus the relevant ones. Irrelevant executions create unnecessary load to the endpoint and reduce the *effectiveness*.
Relevant executions query executions where a change could be detected compared to the last execution, i. e. there was at least one result change since the execution; if there was more than a single change, these updates are counted as missed.
Effectivity the ratio of relevant query executions to total executions.
Absolute delay time between the optimal and actual re-execution (q, i), summed over all queries, which allows to measure the overall *efficiency* of the scheduling strategy.
Maximum delay the longest delay for an individual query execution determines the maximum out-of-date time to be expected from the scheduling strategy for an individual query result. Overly long out-of-date times indicate a *starvation* problem.
Absolute miss number of changes that are recognized, summed over all queries.
Maximum miss the maximum number of missed result updates across all queries.

7 Experimental Results

We have conducted the experiment for three different values of $K_{\mathsf{maxExecTime}}$: 10 sec, 50 sec, and 1,000 sec. This variation of the upper bound execution time allows us to pretend different workloads: As we assume a fixed one-hour interval stepping with 10,000 queries, the workload can be scaled in terms of the number of queries and the time interval, respectively. In the following we present the results for each configuration. The metrics as introduced in Sec. 6 are listed in tabular form.

Table 1 shows the results for $K_{\mathsf{maxExecTime}} = 1{,}000$ sec, which, for our query set, is equivalent to unlimited runtime; that is, all queries could be executed for every revision.

Consequently, the theoretically optimal CV policy has no misses and delay, and executes only relevant queries. In contrast, as the non-selective scheduling policies (RR/ SJF/LJF/CR/DJ) execute all queries and therefore detect all relevant changes, they execute a massive amount of irrelevant queries as overhead, resulting in a low effectivity.

The selective TTL policy reduces the number of query executions effectively, and more updates are detected by resetting a query's time-to-live when a change has been

detected. The best performing configuration tested ($\text{TTL}_{max=32,reset}$) detects 81 % of all changes (12,311 of 15,256) while performing only 3.4 % of the query executions compared to the non-selective policies (738,566 vs. 22,064,744). And still, $\text{TTL}_{max=256}$ detects 75 % (11,459) with 0.75 % query executions (154,185). The reduced query overhead comes at the expense of more delay and in particular higher maximum delay times.

For a runtime limitation of 50 seconds[6], which corresponds roughly to the maximum runtime needed for executing all relevant queries of the query set (cf. Sec. 5), the CV policy has no miss, but cannot execute all queries on time; instead, it delays three relevant executions for one revision each. As expected, SJF has most and LJF has least query executions given an execution time limitation, because short respectively long running queries are preferred. As the decay factor λ is increased, in both cases the number of executed queries tends towards RR. Nevertheless, none of both strategies outperforms RR regarding relevant query executions, delay, or number of misses. The change rate based policies (CR) show that the result history is a good indicator and 92.9 % of changes were detected by $\text{CR}_{\lambda=0.0}$ and 66.7 % by $\text{CR}_{\lambda=0.5}$. The dynamicity-based policy (DJ) detects by far the most result updates (99.7 %) and produces the least delay; the effectiveness is above CR. The TTL configurations show comparable results to the 1000 seconds runtime limitation, i. e. the number of total query executions, detected changes, and the delay remain stable with the 50 seconds limit. Again, we see most result updates are detected by the $\text{TTL}_{max=32,reset}$ configuration.

By looking on the results for the most restrictive execution time limit of 10 seconds in Table 2, we observe that even an optimal scheduling algorithm is not able to detect all result updates in the dataset anymore: the CV policy misses 722 query updates.

LJF closely outperforms RR regarding update detection. RR again has the smallest maximum delay per query. SJF is worse than both LJF and RR in all aspects. The change-based policy (CR) detects updates more effectively. Without decay ($\lambda = 0.0$) it happens that queries that did not change so far are executed very rarely, which results in high delays. Since the maximum miss is relatively high and the total miss is low, we infer that only a small number of frequently changing queries is affected.

The dynamicity-based policy (DJ) detects relatively many updates without executing too many irrelevant queries and, thus, is most effective for the scarce time limita-

[6] For the complete evaluation results we refer to the extended version of this paper [7].

Table 1. Config $K_{\mathsf{maxExecTime}} = 1000sec$

	total qe	irrelevant	relevant	eff. (%)	abs delay	max delay	abs miss	max miss
CV	15,256	0	15,256	100	0	0	0	0
RR/SJF/LJF/CR/DJ	22,080,000	22,064,744	15,256	.07	0	0	0	0
$\text{TTL}_{max=32}$	744,565	732,685	11,880	1.60	26,866	31	3,376	19
$\text{TTL}_{max=32,reset}$	750,877	738,566	12,311	1.64	23,492	31	2,945	19
$\text{TTL}_{max=64}$	405,175	393,507	11,668	2.88	40,747	63	3,588	19
$\text{TTL}_{max=128}$	245,246	233,683	11,563	4.71	61,639	127	3,693	19
$\text{TTL}_{max=128,reset}$	252,714	240,550	12,164	4.81	53,655	127	3,092	19
$\text{TTL}_{max=256}$	165,644	154,185	11,459	6.92	86,202	255	3,797	19

Table 2. Config $K_{\mathsf{maxExecTime}} = 10sec$

	total qe	irrelevant	relevant	eff. (%)	abs delay	max delay	abs miss	max miss
CV	14,484	0	14,484	100	2,481	2	772	2
$\text{LJF}_{\lambda=0.5}$	690,086	687,542	2,544	.37	45,619	35	12,712	31
RR	865,105	862,632	2,473	.29	43,097	31	12,783	28
$\text{SJF}_{\lambda=0.5}$	1,001,825	999,526	2,299	.23	43,498	38	12,957	34
$\text{CR}_{\lambda=0.0}$	109,715	99,791	9,924	9.05	152,346	678	5,332	210
$\text{CR}_{\lambda=0.5}$	676,868	671,640	5,228	.77	45,489	58	10,028	46
DJ	17,519	11,363	6,156	35.1	499,860	2,206	9,100	1750
$\text{TTL}_{max=32}$	621,510	615,662	5,848	.94	37,332	39	9,408	15
$\text{TTL}_{max=256}$	162,407	152,767	9,640	5.94	95,893	258	5,574	18

tion. Nevertheless, this policy is not starvation-free; it ignores queries with less updates. Due to the low dynamicity measure they reach at some point, they henceforth receive a very low rank and are not executed anymore. In contrast, queries with more frequently changing results are preferred and get executed repeatedly. The policy actually only selected 6,282 queries[7] from the query set in total, which indicates a cold start problem. As a result, both the maximum delay and the maximum miss rise significantly.

The TTL policies produce high detection rates for short runtime limitations as well. The maximum delay grows with the maximum time-to-live and $max = 32$ gives the lowest total delays. It can be seen that more changes are detected with a larger time-to-live, but this comes at the cost of delayed update recognition. The maximum numbers of missed updates are low for all TTL configurations compared to the other policies, even though the delay increases.

8 Conclusions

This paper investigates multiple performance metrics of scheduling strategies for the re-execution of queries on a dynamic dataset. The experiments use query results gathered from a large corpus of SPARQL queries executed at more than 2,000 time points of the DBpedia Live dataset, which covers a period of three months. The data collected in the experiments has been made public for comparison with other scheduling approaches.

From the experimental results we conclude that there is no absolute winner. The execution-time-based policies, *Longest-Job-First* and *Shortest-Job-First*, are not able to compete. Compared to *Round-Robin* they generally perform worse. The main advantage of *Round-Robin*, besides its simplicity, is the constantly short maximum delay, but in any setting it can not convince regarding total delay and change detection. *Change-Rate* is able to detect a fair amount of changes. An aging factor should be used under scarce execution time restrictions to prevent long delays. Assuming a limited execution time, the *Dynamics-Jaccard* policy shows best change recognition rates. The effectiveness of this policy as shown in prior work can be confirmed by our results. But, as the

[7] The number of distinct executed queries is not shown in the table, since it is usually 10,000 for all policies except CV.

execution time limit becomes shorter, this policy tends to disregard queries with low update frequencies. Therefore, it is also not starvation-free. As Dividino [1] considered only four iterations, the update frequency of less frequently updated resources could not be measured, but is likely to happen in the dataset update scenario as well. The *Time-To-Live* policy shows a good performance for update detection and can be well adjusted to a certain maximum delay. It keeps the number of maximum missed changes constant. The alternative configuration to reset the time-to-live value instead of dividing it in half when a change has been detected, proves a better performance and results in higher detection rates and also in reduced delays.

It could be shown, that scheduling strategies based on previously observed changes produce better predictions. The *Time-To-Live* policy can be well adapted to required response times. While the *Change-Rate* and *Dynamics* policies proved to detect most updates, they tend to neglect less frequently changing queries. Given a less strict execution time limit, *Dynamics-Jaccard* is the best candidate, else *Time-To-Live* can be recommended because it is starvation-free. For future applications it seems reasonable to combine these scheduling approaches into a hybrid scheduler.

References

1. Dividino, R., Gottron, T., Scherp, A.: Strategies for efficiently keeping local linked open data caches up-to-date. In: The Semantic Web - ISWC 2015, pp. 356–373. Springer (2015)
2. Endris, K.M., Faisal, S., Orlandi, F., Auer, S., Scerri, S.: Interest-based RDF update propagation. In: The Semantic Web-ISWC 2015, pp. 513–529. Springer (2015)
3. Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., Tomasic, A.: Scalable query result caching for web applications. Proc. of the VLDB Endowment 1(1) (2008)
4. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation (2013), https://www.w3.org/TR/sparql11-query/
5. Hellmann, S., Stadler, C., Lehmann, J., Auer, S.: DBpedia Live extraction. In: On the Move to Meaningful Internet Systems: OTM 2009, vol. 5871, pp. 1209–1223. Springer (2009)
6. Kjernsmo, K.: A survey of http caching implementations on the open semantic web. In: The Semantic Web. Latest Advances and New Domains. pp. 286–301. Springer (2015)
7. Knuth, M., Hartig, O., Sack, H.: Scheduling Refresh Queries for Keeping Results from a SPARQL Endpoint Up-to-Date (Extended Version). CoRR abs/1608.08130 (2016)
8. Knuth, M., Reddy, D., Dimou, A., Vahdati, S., Kastrinakis, G.: Towards linked data update notifications - reviewing and generalizing the sparqlPuSH approach. In: Proc. NoISE (2015)
9. Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing linked data dynamics. In: The Semantic Web: Semantics and Big Data: ESWC. Springer (2013)
10. Martin, M., Unbehauen, J., Auer, S.: Improving the performance of semantic web applications with SPARQL query caching. In: Proc. of ESWC. Springer (2010)
11. Passant, A., Mendes, P.N.: sparqlPuSH: Proactive notification of data updates in RDF stores using PubSubHubbub. In: Proc. of Scripting for the Semantic Web Workshop (2010)
12. Popitsch, N., Haslhofer, B.: DSNotify–a solution for event detection and link maintenance in dynamic datasets. Journal of Web Semantics 9(3), 266–283 (2011)
13. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.C.N.: LSQ: The linked SPARQL queries dataset. In: The Semantic Web - ISWC 2015. Springer (2015)
14. Tramp, S., Frischmuth, P., Ermilov, T., Auer, S.: Weaving a social data web with semantic pingback. In: Knowledge Engineering and Management by the Masses. Springer (2010)
15. Williams, G.T., Weaver, J.: Enabling fine-grained HTTP caching of SPARQL query results. In: The Semantic Web–ISWC 2011, pp. 762–777. Springer (2011)