Predicting SPARQL Query Dynamics

Alberto Moya Loustaunau
IMFD; DCC, University of Chile
Santiago, Chile
amoya@dcc.uchile.cl

Aidan Hogan
IMFD; DCC, University of Chile
Santiago, Chile
ahogan@dcc.uchile.cl

ABSTRACT

Given historical versions of an RDF graph, we propose and compare several methods to predict whether or not the results of a SPARQL query will change for the next version. Unsurprisingly, we find that the best results for this task are achievable by considering the full history of results for the query over previous versions of the graph. However, given a previously unseen query, producing historical results requires costly offline maintenance of previous versions of the data, and costly online computation of the query results over these previous versions. This prompts us to explore more lightweight alternatives that rely on features computed from the query and statistical summaries of historical versions of the graph. We evaluate the quality of the predictions produced over weekly snapshots of Wikidata and daily snapshots of DBpedia. Our results provide insights into the trade-offs for predicting SPARQL query dynamics, where we find that a detailed history of changes for a query's results enables much more accurate predictions, but has higher overhead versus more lightweight alternatives.

CCS CONCEPTS

• **Information systems** → **Temporal data**; **Resource Description Framework (RDF)**; **Query languages**.

KEYWORDS

Dynamics, Linked Data, SPARQL, RDF

ACM Reference Format:
Alberto Moya Loustaunau and Aidan Hogan. 2021. Predicting SPARQL Query Dynamics. In *Proceedings of the 11th Knowledge Capture Conference (K-CAP '21), December 2–3, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3460210.3493565

1 INTRODUCTION

Recent years have seen increased interest in querying graphs, particularly in the context of NoSQL systems, Linked Data, Knowledge Graphs, etc. A prominent data model for graphs is the Resource Description Framework (RDF), whose standard query language is SPARQL. Hundreds of RDF datasets have been published on the Web following the Linked Data principles. These datasets often provide a SPARQL service (known as an *endpoint*) that clients can query over the Web [2]. The most prominent of these datasets – including the likes of DBpedia [19] and Wikidata [42] – provide public access to structured descriptions of millions of entities and their relations. SPARQL endpoints over such datasets can receive in the order of millions of queries per day from the Web [21, 35].

SPARQL endpoints can suffer from timeouts and intermittent availability [2]. An established way to improve the performance of databases and Web-based systems is to apply server-side and/or client-side caching to reuse some of the work done for previous requests and/or to keep frequently accessed (partial) results in-memory [16, 18, 22, 30, 45]. However, datasets are subject to change over time [14], which may render cached data stale. When deciding what results to cache, it is important to consider how long the results will stay valid. On the server-side, caching dynamic results that will soon become stale can be a waste of resources. On the client-side, refreshing cached results by resending queries to the server is necessary to avoid stale data, but frequent refresh queries can put an unnecessary load on the endpoint's server [5, 15, 16, 29].[1] Though works have studied the dynamics of various aspects of RDF graphs and of Linked Data [6, 11, 14, 29], we see that more work is needed on predicting the dynamics of SPARQL query results.

Contributions: In previous work [25], we proposed a machine learning method to predict whether or not a SPARQL query's results will change in the next version of a dynamic RDF graph based on features computed from (1) the query itself, (2) statistics of the dynamicity of the query's predicates, and (3) query results over historical versions. In experiments over Wikidata, we found that features based on historical results provided by far the most accurate predictions. However, such features require evaluating previously-unseen queries over multiple versions at runtime, incurring too high an overhead for many use-cases (e.g., for caching, whose goal is to *reduce* runtime overhead). In this paper, we propose novel features based on estimates of the *degree of change*, i.e., the ratio of results that change between versions. These estimates are extracted from high-level statistics and do not require historical query results. We compare the predictions made by the feature sets with respect to two real-world benchmarks that evaluate queries over 17 weekly versions of Wikidata [42] and 18 daily versions of DBpedia Live [24].

Hypothesis: We hypothesise that the methods we propose follow a trade-off for predictions, as illustrated in Figure 1, where more accurate predictions imply greater overheads. On the left-hand side, we have query features, which require no knowledge of the data, and thus involve the least overhead, but provide less accurate predictions as a result. Next we have features about the dynamics of predicates used in the query, which require high-level statistics about how the triples involving a particular predicate change.

K-CAP '21, December 2–3, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8457-5/21/12…$15.00
https://doi.org/10.1145/3460210.3493565

[1] An alternative is to apply a *push-based model*, where the server notifies clients of changes [17, 31], but this becomes untenable when dealing with thousands or millions of clients. Knuth et al. [17] discuss further limitations of this approach.

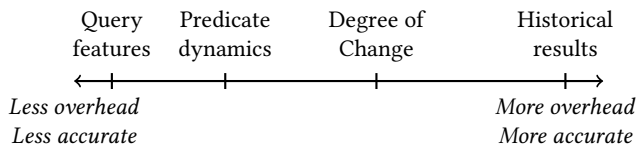

Figure 1: Hypothesised trade-off for predicting the dynamics of the results of unseen queries

Thereafter, the degree-of-change measures require more detailed statistics, but allow for computing more detailed meta-data regarding a particular query's sensitivity to changes in the graph. Finally, we have knowledge of historical results, which for an unseen query, requires evaluating the query on several historical versions, and maintaining indexes over those versions, thus implying a much higher overhead, but with the benefit of having much more detailed information about how its results have changed over past versions. Concretely, we hypothesise that the accuracy and overhead of predictions using different feature sets follows the ordering shown in Figure 1. If this hypothesis is validated, it would imply that there is no "one size fits all" solution to such prediction, but rather that the choice of features on which to base the prediction depends on the priorities of efficiency vs. accuracy in the context of the use-case.

Paper structure: Section 2 gives preliminaries for RDF/SPARQL. Section 3 then provides a formal statement of the problem we address. Section 4 discusses related works. Section 5 describes our proposed approach for predicting query dynamics. Section 6 presents the design and results of the experiments. We conclude and review limitations and future directions in Section 7.

2 PRELIMINARIES

We begin by defining preliminaries related to RDF and SPARQL.

RDF is a graph-based data model recommended for use on the Web. RDF is based on three pairwise-disjoint sets of *RDF terms*: IRIs $\mathbf{I}$, literals $\mathbf{L}$, and blank nodes $\mathbf{B}$. An RDF *triple* $(s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times (\mathbf{I}) \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is composed of a *subject*, a *predicate*, and an *object*. An *RDF graph* is a set of RDF triples.

SPARQL is the query language recommended for use with RDF. In this work, we focus on `SELECT` queries using core features from SPARQL 1.0, namely basic graph patterns, unions, optionals, projection and distinct. Further query features can be supported as an extension of our proposed method, left for future work.

Queries introduce a fourth set of terms: variables from the set $\mathbf{V}$ that are disjoint with IRIs, literals and blank nodes. A *triple pattern* $t \in (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{B} \cup \mathbf{V})$ is then an RDF triple that allows variables in any position. A *basic graph pattern* is a set of triple patterns. For simplicity, we assume that blank nodes in triple patterns are rewritten to fresh variables that are projected away. We currently do not support blank nodes in the solutions of a query, but they could be handled in future by skolemising them in the input graph, or by canonically labelling them in the solutions [13].

A SPARQL *query pattern* is built recursively as follows:

(1) A *basic graph pattern* B is a *query pattern*.
(2) If Q_1 and Q_2 are query patterns, then Q_1 `AND` Q_2, Q_1 `UNION` Q_2, and Q_1 `OPTIONAL` Q_2, are query patterns.
(3) Finally, if Q is a query pattern, V a list of variables and Δ a boolean value, $\texttt{SELECT}^{\Delta}_{V} Q$ is a SPARQL `SELECT` query, where V denotes the projected variables, and Δ the `DISTINCT` option that when true, removes duplicate results.

The semantics of a SPARQL `SELECT` query Q is defined in terms of its *evaluation* over an RDF graph G, denoted $Q(G)$, which returns a bag (if `DISTINCT` is omitted) or a set (if `DISTINCT` is included) of solution mappings. We assume here the standard definitions for the evaluation of such queries (see, e.g., Pérez et al. [32]), where basic graph pattern matching is homomorphism-based, `AND` is evaluated as a natural join ($\bowtie$), `UNION` as a bag union ($\cup$), `OPTIONAL` as a natural left-join ($\rtimes$) and `SELECT` as projection (π).

3 PROBLEM STATEMENT

We now formally state the problem that we address. We consider a *dynamic RDF graph* to be an RDF graph that changes over time. We represent a dynamic RDF graph $\mathcal{G} = (G_1, \ldots, G_n)$ as a tuple of n RDF graphs, where each RDF graph denotes a discrete version. In the prediction setting, latter versions may be future/unknown at a given point in time; as a convention, we will use k to index the current version in $\mathcal{G} = (G_1, \ldots, G_k, G_{k+1}, \ldots, G_n)$ where $1 \leq k < n \leq \infty$. Given a query and a dynamic RDF graph, the central problem that interests us in this paper is as follows:

One-Shot-Change (OSC)
INPUT: a dynamic RDF graph $\mathcal{G} = (G_1, \ldots, G_k)$; a SPARQL query Q.
OUTPUT: `true` if $Q(G_k) \neq Q(G_{k+1})$ is predicted; `false` otherwise.

Note that by $Q(G) = Q(G')$, we refer to bag equality, where order does not matter but multiplicities are taken into account.

Example 3.1. Take the dynamic RDF graph $\mathcal{G} = (G_1, G_2, G_3, \ldots)$ with three known versions of Wikidata graphs shown in Figure 2.[2] Between G_1 and G_2, the name of Swaziland is changed to Eswatini and the old name is added as an alias. Between G_2 and G_3, the claim that Luke Hall has the occupation of swimmer is changed to state that his sport is swimming. Given a query Q such as the following, which looks for names of swimmers and their countries:

```
SELECT DISTINCT ?swimmerName ?countryName {
  ?swimmer rdfs:label ?swimmerName .
  ?swimmer wdt:P106 wd:Q10843402 . # occupation swimmer
  ?swimmer wdt:P27 ?country .      # nationality
  ?country rdfs:label ?countryName .
}
```

the goal of OSC is then to predict whether or not the results for $Q(G_4)$ will change with respect to $Q(G_3)$. □

In terms of desiderata, the procedure for this problem should provide accurate predictions (which we can evaluate by passing a prefix of known versions, making predictions for held-out versions). Rather than requiring the full dynamic RDF graph $\mathcal{G}$ as input, ideally the procedure would accept $\mathrm{d}(\mathcal{G})$ and Q as input, where $\mathrm{d}(\mathcal{G})$ is a description of the dynamic graph that is computed from $\mathcal{G}$ but is (far) more concise than $\mathcal{G}$. A server wishing to implement OSC would then have less data to manage and could even publish $\mathrm{d}(\mathcal{G})$

[2] The prefixes used in this paper are as defined on http://prefix.cc

Legend	wdt:P27/nationality wdt:P31/instance of wdt:P106/occupation wdt:P641/sport wd:Q1050/Eswatini/Swaziland wd:Q31920/swimming wd:Q2686854/Luke Hall wd:Q3624078/sovereign state wd:Q10843402/swimmer		
Version	G_1	G_2	G_3
Triples	wd:Q1050 rdfs:label "Swaziland"@en . wd:Q1050 wdt:P31 wd:Q3624078 . wd:Q2686854 rdfs:label "Luke Hall"@en . wd:Q2686854 wdt:P27 wd:Q1050 . wd:Q2686854 wdt:P106 wd:Q10843402 .	wd:Q1050 rdfs:label "Eswatini"@en . wd:Q1050 skos:altLabel "Swaziland"@en . wd:Q1050 wdt:P31 wd:Q3624078 . wd:Q2686854 rdfs:label "Luke Hall"@en . wd:Q2686854 wdt:P27 wd:Q1050 . wd:Q2686854 wdt:P106 wd:Q10843402 .	wd:Q1050 rdfs:label "Eswatini"@en . wd:Q1050 skos:altLabel "Swaziland"@en . wd:Q1050 wdt:P31 wd:Q3624078 . wd:Q2686854 rdfs:label "Luke Hall"@en . wd:Q2686854 wdt:P27 wd:Q1050 . wd:Q2686854 wdt:P641 wd:Q31920 .
Added		wd:Q1050 rdfs:label "Eswatini"@en . wd:Q1050 skos:altLabel "Swaziland"@en .	wd:Q2686854 wdt:P641 wd:Q31920 .
Removed		wd:Q1050 rdfs:label "Swaziland"@en .	wd:Q2686854 wdt:P106 wd:Q10843402 .

Figure 2: Example of a dynamic RDF graph $\mathcal{G} = (G_1, G_2, G_3, \ldots)$ with three known versions based on Wikidata

so that clients can independently predict when results will change. For example, d($\mathcal{G}$) might be a statistical summary of the graphs in $\mathcal{G}$ and what changes between its versions.

Example 3.2. Consider the query Q from Example 3.1 and the dynamic RDF graph $\mathcal{G} = (G_1, G_2, G_3, \ldots)$ from Figure 2. In order to implement OSC, a natural strategy would be to evaluate and compare $Q(G_1)$ with $Q(G_2)$ and $Q(G_2)$ with $Q(G_3)$. Since the results change in both pairs of versions, we may say that it is likely they will change again between $Q(G_3)$ and the future version $Q(G_4)$. However, evaluating $Q(G_1)$, $Q(G_2)$ and $Q(G_3)$ requires maintaining the full versioning history, which may be costly. Also, if Q is a previously unseen query, we incur the runtime cost of evaluating $Q(G_1)$, $Q(G_2)$ and $Q(G_3)$. This approach is thus costly.

Alternatively, we may look at other clues for OSC that do not require the costly evaluation of $Q(G_1)$, $Q(G_2)$ and $Q(G_3)$. First, we may note that the query has four variables and two projected variables with distinct; we might expect a query with more variables (particularly projected variables) to be more sensitive to change, a query with distinct to be less sensitive to change, etc.; hence we propose that *query features* may potentially be useful for OSC. Second, we may note that the query mentions two predicates (`rdfs:label` and `wdt:P106`) whose triples are sometimes involved in changes between historical versions, which may suggest that the query is more sensitive to change; hence we propose that *predicate dynamicity features* may be useful for OSC. Finally, we may use statistical summaries of (G_1, G_2, G_3) to estimate and compare the cardinality of $Q(G_1 \cap G_2)$ and $Q(G_1 \cup G_2)$, as well as $Q(G_2 \cap G_3)$ and $Q(G_2 \cup G_3)$, without evaluating these queries, giving us an estimate of what we call the *degree of change* in the results of Q – i.e. the ratio of results for Q that are sensitive to changes – across pairs of versions, which we again propose may be useful for the OSC task.

Given the diversity of such clues that may influence different aspects of OSC predictions, and the potential interactions between them, we propose to encode these clues as features and use them to train a binary classifier for computing the final prediction. □

Though we focus on OSC, the Time To Live (TTL) task, which predicts the next version m ($m > k$) in which the results $Q(G_m)$ are expected to change (if any), can be addressed by applying OSC with gaps between versions. Given a query Q and a dynamic RDF graph $\mathcal{G} = (G_1, \ldots, G_k)$, we can call OSC with $\mathcal{G}$ to predict a change for $Q(G_{k+1})$; if true, we return $k + 1$; if false, we can use $\mathcal{G}_2 = (G_1, G_3, \ldots, G_{k-\text{mod}(k-1,2)})$, returning $k + 2$ if OSC returns true, or moving onto an interval of three if it returns false, and so forth, outputting the lowest value $j < k$ such that OSC on $\mathcal{G} = (G_1, G_{j+1}, \ldots, G_{k-\text{mod}(k-1,j)})$ and Q returns true, or returning that there is no foreseeable change if no such value for j is found.[3]

4 RELATED WORK

Before presenting our proposals for implementing OSC, we discuss works on RDF/SPARQL dynamics. We also briefly discuss works in the relational database literature regarding SQL.

RDF/Linked Data dynamics refers to how RDF graphs (on the Web) change over time, and has been studied from numerous perspectives. While some works have focused on studying changes over time within a particular RDF graph [11], others have focused on changes across different datasets [7, 8, 14, 28, 38, 40], including new datasets being added or removed [14, 40]. Dynamics have been studied at the level of entities [11, 28, 40], predicates [14], schemata [8], documents [7, 14, 38, 40] and entire datasets [14]. Many of these works have been observational in nature, trying to capture and analyse patterns in dynamic Linked Data content [7, 8, 14, 28, 38]. Such works have also proposed models to characterise dynamics – and ultimately predict changes – based on Poisson distributions [38], abstract schemata [8], Markov chains [41], empirical distributions [27], machine learning models [29], formal concept analysis [11], among other techniques.

Synchronisation refers to the problem of keeping multiple clients up-to-date with remote changes on a server as efficiently (for both client and server) as possible. There are two main approaches to synchronisation. *Push-based* approaches are typically based on servers notifying clients of relevant changes [10, 17, 23, 31, 37]. Such approaches enable stronger levels of consistency, but centralise the burden of synchronisation on the server. Conversely, in *pull-based approaches*, the responsibility lies with clients to request updated data from the server [1, 7, 15, 16]. Such approaches involve weaker levels of consistency, but preserve the traditional style of client–server interaction that has enabled the Web to scale.

[3] More advanced reductions from TTL to OSC might consider applying binary search to find j; or using a voting scheme over (up to) $j - 1$ dynamic graphs with gaps of j, such as $(G_1, G_4, \ldots)$, $(G_2, G_5, \ldots)$, $(G_3, G_6, \ldots)$ for $j = 3$, etc.

SPARQL query caching reuses the results of SPARQL (sub-)queries for subsequent requests. Numerous server-side caching techniques for SPARQL have been proposed, based on similarity measures for graph patterns [20, 36], the types of join used [18], generalised and canonicalised sub-queries [30, 34, 44], etc. Rather than caching (sub-)query results, Zhang et al. [45] cache frequently accessed triples. Only the works of Martin et al. [22] and Williams and Weaver [43] consider changes in the RDF graph; the former work invalidates cached results for a query when a triple matching one of its query patterns changes, while the latter work adds modification times to the RDF index to quickly detect changes. On the client-side, Knuth et al. [16] propose a service in which clients can register queries that are scheduled for refresh according to policies such as how long ago the last refresh took place, how long the query takes to run, how often the results change, how many results change, etc.

Caching in relational databases has been studied under topics such as *semantic caching* in client–server settings, where a client maintains semantic descriptions of cached data that facilitates their future reuse [3, 33]; *incremental view maintenance* [12], where a materialised view, storing the results of a query, can be updated to reflect changes in the underlying data rather than being recomputing from scratch; etc. We are not, however, aware of works from the relational literature that try to *predict* changes in query results.

Novelty: We aim to predict whether or not the results of a SPARQL query will change in the next version of a dynamic RDF graph. Previous works in this direction have been primarily based on dynamic predicate features [4, 39] and/or historical results for the query [16]. We consider these methods, where we find that predicate features, though more lightweight, offer poorer prediction accuracy. We thus propose novel features based on estimating the degree of change using cardinality estimations in order to strike a novel balance between efficiency and accuracy. Our work complements works on caching and synchronisation, which we see as use-cases: our method can be used, for example, by a server to decide for which queries to cache results, or by a client to decide when to refresh the results for a given query by evaluating it again remotely.

5 PREDICTING OSC QUERY DYNAMICS

We now describe our approach for predicting changes in the results of a particular query. Given a dynamic RDF graph and a query, the general idea is to extract a set of features for the query relative to the dynamic RDF graph that can be used for the purposes of binary classification, predicting an answer for the OSC task.

5.1 Architecture

We present the architecture for our proposed system for predicting OSC in Figure 3. The inputs are a query Q and a dynamic RDF graph $\mathcal{G}$. The system then extracts a feature vector $(f_1, \ldots, f_k)$. from these inputs and feeds them into a pre-trained binary classifier to make the OSC prediction. The query features $(f_1, \ldots, f_i)$ are extracted online from the query itself. The predicate and degree-of-change features $(f_{i+1}, \ldots, f_j)$ are extracted from a statistical description $\mathrm{d}(\mathcal{G})$ of $\mathcal{G}$, whose details will be described later; in practice, $\mathrm{d}(\mathcal{G})$ can be computed and maintained offline (independently of the query) in an incremental manner, requiring only the two most

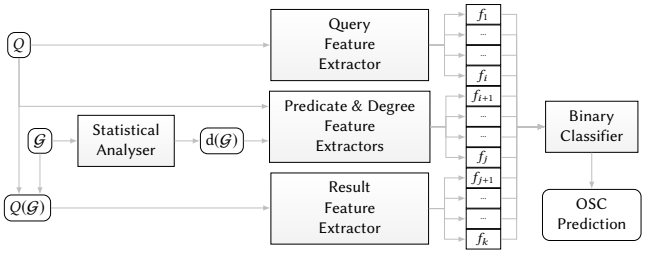

Figure 3: Proposed architecture for predicting change OSC given a query Q and dynamic RDF graph $\mathcal{G}$

recent versions of $\mathcal{G}$ to be updated. Finally, the results features $(f_{j+1}, \ldots, f_k)$ require as input the full historical results of Q for each version of $\mathcal{G}$ (which we denote by $Q(\mathcal{G})$); this must be computed online. The binary classifier is pre-trained over a given set of queries $\mathcal{Q}$ for which ground truths are computed over withheld versions.

5.2 Features

We now present the details of our features. The degree-of-change features are new to this work, while query, predicate, and results features were also explored in our previous work [25].

Query features (Q*).* include statistics about the query. Formally, given a query Q, we define that $\mathrm{Q}(Q) = (n_T, n_V, n'_V, \vec{o})$, where n_T denotes the number of triple patterns in Q, n_V denotes the number of variables in Q, n'_V denotes the number of projected variables in Q, and $\vec{o}$ is a one-hot encoded vector that denotes the SPARQL operators (`OPTIONAL`, `UNION`, etc.) used by Q. We hypothesise that such features may be useful for predicting dynamics, where triples with fewer triple patterns, variables, etc., will have fewer "opportunities" to be affected by changes in query results. However, there is no guarantee that query dynamics will correlate with them; for example, additional triple patterns may constrain the results of a query by making it more specific, and thus making its results less likely to change. The value of query features for predicting dynamics will need to be studied empirically.

Predicate features (P*):* include statistics about how frequently and how many triples matching the predicates used in the query change. More formally, let $G_1 \oplus G_2 = G_1 \setminus G_2 \cup G_2 \setminus G_1$ denote the set of triples in one graph, but not in the other. Further let $\sigma_{\mathrm{P}=p}(G) = \{(x, y, z) \in G \mid p = y\}$ denote the triples in G using the predicate p. Now, given a dynamic RDF graph $\mathcal{G} = (G_1, \ldots, G_k)$, we denote by $\Delta(\mathcal{G}, p) = \sum_{i=1}^{k-1} \frac{|\sigma_{\mathrm{P}=p}(G_i \oplus G_{i+1})|}{|\sigma_{\mathrm{P}=p}(G_i \cup G_{i+1})|}$ the sum of the ratios of triples for p that changed between each consecutive pair of versions, such that the higher the value for $\Delta(\mathcal{G}, p)$, the more dynamic the triples associated with the predicate. Next let $\mathrm{preds}(Q)$ denote the set of IRIs used as predicates in Q. Now we define $\mathrm{P}(\mathcal{G}, Q) = \frac{\sum_{p \in \mathrm{preds}(Q)} \Delta(\mathcal{G}, p)}{|\mathrm{preds}(Q)|}$, taking the mean value of $\Delta(\mathcal{G}, p)$ for all predicates in Q. Predicate features only require lightweight statistics about the dynamic RDF graph since the number of unique predicates – even in large RDF graphs – tends to be relatively small. However these features take into account few details of the query; for example, a query searching for all labels and a query searching for the label of a specific subject will not be distinguished.

*Degree-of-change features (*D*):* include statistics about the variability in the number of results returned by the query across the historical versions. Specifically, given a query Q and an RDF graph G, we denote by $\text{card}(\cdot,\cdot)$ a *cardinality estimation function*, where $\text{card}(Q,G)$ estimates the (bag) cardinality of $|Q(G)|$. Any such function can be used; we describe the one we use in Section 5.4. We use the function $\text{card}(Q,G)$ to estimate the dynamics of a query by comparing the number of results for the query that depend on data that did not change between two versions, versus the number of results generated over the union of the two versions. Specifically, given a dynamic RDF graph $\mathcal{G} = (G_1,\ldots,G_k)$, we sum the ratios $\text{D}(\mathcal{G},Q) = \sum_{i=1}^{k-1} 1 - \frac{\text{card}(Q,G_i \cap G_{i+1})}{\text{card}(Q,G_i \cup G_{i+1})}$ to estimate the degree-of-change, i.e., the ratio of query results that are sensitive to changes between versions.[4] When compared with query features, cardinality features take into account changes in the historical versions of the graph. When compared with predicate features, degree-of-change features take into account more details of the query, but require additional statistics in order to estimate cardinalities.

*Results features (*R*):* include statistics about the historical results for the query. These features are conceptually the simplest, where we simply count the number of times the results for the query Q changed between the pairs of consecutive versions of the dynamic RDF graph. Formally, given a query Q and a dynamic RDF graph with k known versions $\mathcal{G} = (G_1,\ldots,G_k)$ we define this feature as: $\text{R}(\mathcal{G},Q) = |\{i \in \{1,\ldots,k-1\} : Q(G_i) \neq Q(G_{i+1})\}|$. Though conceptually the simplest, we expect that this feature will also provide the most useful information for OSC prediction. Conversely, it incurs the highest overhead for an unseen query Q, requiring maintaining all data for $G_1,\ldots,G_k$ offline and evaluating the queries used for training over these versions, as well as evaluating $Q(G_1),\ldots,Q(G_k)$ online (when the query is received).

5.3 Dynamic graph description

In the case of query features, we do not require direct knowledge of the dynamic graph. In the case of predicate and degree-of-change features, while we require some knowledge of the dynamic graph, we can also compute these features from a dynamic graph description that provides key statistics relating to a graph.

Specifically, given an RDF graph G, we define its key statistics as a tuple $\text{s}(G) = (n_T, n_S, n_P, n_O)$ where $n_T = |G|$ denotes the number of triples in G, $n_S := |\{x : \exists y,z\ (x,y,z) \in G\}|$ denotes the number of unique subjects in G, $n_P := |\{y : \exists x,z\ (x,y,z) \in G\}|$ denotes the number of unique predicates in G, while, analogously, $n_O := |\{z : \exists x,y\ (x,y,z) \in G\}|$ denotes the number of unique objects in G. By $\text{d}(G) = (\text{s}(G), \{(p_1, \text{s}(\sigma_{\text{P}=p_1}(G))),\ldots,(p_n, \text{s}(\sigma_{\text{P}=p_k}(G)))\})$ we denote the description of G, consisting of the key statistics of G, and the key statistics for the partitions of G by predicate, where $\text{preds}(G) = \{p_1,\ldots,p_k\}$.[5] For brevity, we will write $n_T[p]$, $n_S[p]$ and $n_O[p]$ to denote the values for n_T, n_S and n_O in $\text{s}(\sigma_{\text{P}=p}(G))$, i.e., the number of unique triples, subject and objects, respectively for the sub-graph of G with predicate p; we will similarly write n_T, n_S and n_O to more concisely denote the values in $\text{s}(G)$.

We can now generalise this idea to a dynamic graph description. Given a dynamic graph $\mathcal{G} = (G_1,\ldots,G_k)$, we compute $6(k-1)$ graph descriptions, or in other words two graph descriptions for each adjacent pair of graphs G_i, G_{i+1}, specifically, $\text{d}(G_i \cap G_{i+1})$, $\text{d}(G_i \oplus G_{i+1})$, $\text{d}(G_i \cup G_{i+1})$. Rather than describing each individual graph, this allows us to capture and compare statistics about the triples that stayed the same in both versions, that changed between versions, and that appeared in either version, respectively. A dynamic graph description $\text{d}(\mathcal{G})$ is then a 3-tuple of the form:

$$\begin{aligned}\text{d}(\mathcal{G}) = ((&\text{d}(G_1 \cap G_2),\ldots,\text{d}(G_{k-1} \cap G_k))\\ &(\text{d}(G_1 \oplus G_2),\ldots,\text{d}(G_{k-1} \oplus G_k))\\ &(\text{d}(G_1 \cup G_2),\ldots,\text{d}(G_{k-1} \cup G_k)))\end{aligned}$$

The predicate features described earlier are then trivial to compute from $\text{d}(\mathcal{G})$. This leaves us to discuss the estimations of cardinalities needed for the degree-of-change features.

5.4 Cardinality estimations

Any cardinality estimation function $\text{card}(\cdot,\cdot)$ can be used within our method, where the more accurate the function is in terms of predicting the actual number of results returned by Q over G, the better the results we would expect for predicting dynamics. An obvious implementation is to use $\text{card}(Q,G) = |Q(G)|$, i.e., to evaluate the query on the graph and count the results. However, this is akin to using historical query results, and will be costly. We instead define a function $\text{card}(\cdot,\cdot)$ that uses (only) our dynamic-graph description. Specifically, given an RDF graph G, we use $\text{d}(G)$ to estimate the cardinality of triple patterns per standard cardinality estimations used by relational databases (e.g., by Postgres[6]), considering each predicate as a binary relation (M_i). We adopt typical assumptions in these scenarios, such as the assumption that every join is complete, and that (subject/object) values are uniformly distributed.

Let $X,Y,Z \in \mathbf{V}$ and $x,y,z \in \mathbf{I} \cup \mathbf{L}$. We estimate the cardinality of a triple pattern t, based on these assumptions, as follows:

$$\begin{aligned}
&\text{if } t = (x,y,z) \text{ then } \text{card}(t,\text{d}(G)) = 1\\
&\text{if } t = (X,y,z) \text{ then } \text{card}(t,\text{d}(G)) = n_T[y]/n_O[y]\\
&\text{if } t = (x,Y,z) \text{ then } \text{card}(t,\text{d}(G)) = 1\\
&\text{if } t = (x,y,Z) \text{ then } \text{card}(t,\text{d}(G)) = n_T[y]/n_S[y]\\
&\text{if } t = (X,Y,z) \text{ then } \text{card}(t,\text{d}(G)) = n_T/n_O\\
&\text{if } t = (X,y,Z) \text{ then } \text{card}(t,\text{d}(G)) = n_T[y]\\
&\text{if } t = (x,Y,Z) \text{ then } \text{card}(t,\text{d}(G)) = n_T/n_S\\
&\text{if } t = (X,Y,Z) \text{ then } \text{card}(t,\text{d}(G)) = n_T
\end{aligned}$$

The resulting cardinality is only exact for (X,y,Z) and (X,Y,Z). Otherwise, when both subject and object are constant, we assume that the cardinality is 1; when one of subject or object is constant, we assume that the cardinality is a ratio of the total number of triples in the corresponding graph divided by the number of unique subject/object terms in the graph (per a uniform distribution).

The distribution of subject and object values may sometimes be far from uniform. For example, a country such as India may appear much more often as the object of the predicate *birthPlace* than a country such as Tuvalu. We thus extend $\text{s}(G)$ in order to store the κ

[4] While it may be simpler – and thus tempting – to rather compare $\text{card}(Q, G_i)$ and $\text{card}(Q, G_{i+1})$, they may generate the same number of results even though the results themselves change. This issue would arise, for example, if we were to query for the objects of `rdfs:label` over Figure 2, where the number of results per version remains precisely the same even though the results change between G_1 and G_2.

[5] Note that $n_P = 1$ for any $\text{s}(\sigma_{\text{P}=p_i}(G))$ with $1 \leq i \leq k$, and is not stored.

[6] https://www.postgresql.org/docs/current/static/planner-stats-details.html

most common subject and object values and their frequencies, both for the full graph and its predicate partitions. Thus if we have a pattern (x, y, Z), where x is in the top-κ most common subjects for the sub-graph with predicate y, we take the exact cardinality. The estimations for non top-κ values are also adjusted accordingly to subtract the cardinalities of the top-κ values from the calculations.

The estimations for other query operators are then layered on top of the estimates for triple patterns in a standard way. Specifically, for each bag of solution mappings produced at each stage, we maintain an estimated cardinality for the entire bag, as well as an estimated number of unique values for each variable. When we compute the cardinality estimate for a join $M_1 \bowtie M_2$ between two bags of solutions M_1 and M_2 on the variable V, if we estimate that V takes v_1 unique values in M_1 and v_2 unique values in M_2, then we assume that V will have $\min\{v_1, v_2\}$ unique values in $M_1 \bowtie M_2$ based on the assumption of complete joins. Without loss of generality, if we assume that $v_1 \leq v_2$, then the estimated number of unique values for (other) variables in M_2 will be reduced by a factor of $\frac{v_1}{v_2}$. Finally, the number of mappings in $M_1 \bowtie M_2$ will be computed as $\frac{m_1 m_2}{v_2}$, where m_1 and m_2 denote the estimated cardinalities for M_1 and M_2 respectively. We follow a similar process for estimating the cardinalities of the left-join $M_1 \bowtie M_2$, where we do not lower the estimates for unique values of variables in M_1. For the union $M_1 \cup M_2$, we sum (under bag semantics) the estimates for M_1 and M_2 and sum the estimates of unique values per variable (effectively assuming all values to be distinct in M_1 and M_2). For projection combined with distinct, the overall cardinality is given by the maximum number of unique values for an individual projected variable, with each variable maintaining its estimate.

For computing the degree-of-change features, the dynamic-graph description $\mathrm{d}(\mathcal{G})$ contains the individual graph descriptions needed to compute $\mathrm{card}(Q, G_i \cap G_{i+1})$ and $\mathrm{card}(Q, G_i \cup G_{i+1})$, namely $\mathrm{d}(G_i \cap G_{i+1})$ and $\mathrm{d}(G_i \cup G_{i+1})$ for $1 \leq i < k - 1$.

5.5 Classification

To make the final OSC prediction for Q and $\mathcal{G}$, we use a binary classifier trained over example queries. Specifically, given a dynamic RDF graph $\mathcal{G} = (G_1, \ldots, G_k)$ and a set of training queries $\boldsymbol{Q}$, we use a sub-sequence of versions $\mathcal{G}' = (G_i, \ldots, G_j)$ (where $1 \leq i < j < k$) in order to build a feature set for each query $Q \in \boldsymbol{Q}$, and evaluate $Q(G_j) = Q(G_{j+1})$ in order to label the query. Given an unseen query Q', we can then build the feature set with respect to $\mathcal{G}$ (or any sub-sequence thereof if we do not wish to use all versions), and use the trained classifier to predict whether or not the results for Q' change for the next (withheld) version of the graph.

6 EVALUATION

Recalling the hypothesised trade-off shown in Figure 1, we first focus on the accuracy of the features for OSC precision, and then summarise the cost of extracting different features.

6.1 Accuracy

This evaluation aims to ascertain the quality of predictions (in terms of precision, recall and F_1-score) as to whether or not the results for a query will change in the next version considering these features. We begin by describing the two datasets used.

Datasets. We first consider a dataset of 17 weekly "truthy" versions of Wikidata[7], spanning from 2019/11/14 (containing 4.4 billion triples) to 2020/03/05 (containing 4.8 billion triples). The number of triples grew by 9.2% during the time period, where approximately 4 times more triples are added, on average, than removed. A total of 78 billion triples are present in all versions. We use 141 queries that were sourced from the user-contributed example queries published by the Wikidata query service. The results for 20 queries never change, while 56 change between each pair of versions [25].

We consider a second dataset based on DBpedia Live, where we use the changesets to build 18 daily versions, spanning from 2019-07-01 (containing 593.6 million triples), until 2019-07-18 (containing 590.3 million triples). In this case we see more deletions than insertions, and fewer changes overall, when compared with Wikidata. This is to be expected considering that the interval between versions is 7 times shorter in the case of DBpedia. Overall, the DBpedia versions contain 10.7 billion triples. We use the set of 10,000 queries extracted by Knuth et al. [16] from the LSQ dataset [35], composed of queries evaluated by the DBpedia SPARQL endpoint. We filtered these queries to remove those that returned empty results over all versions. However, given the origin of these queries, we noted a very high number of very similar queries, which we assume are due to applications using the endpoints. To avoid the results being skewed by repetitive queries, we partitioned the queries by their predicate set, and applied a logarithmic downsampling to reduce the number of similar queries in each partition such that there were fewer than 20 queries per partition. The end result was 256 queries, where there were 416 changes in results between pairs of versions (out of a possible 4,318 comparisons). Of the 256 queries, 54 have at least one change: 16 queries change each time while 21 change only once. We note fewer changes than in the case of Wikidata, likely due to the narrower gap between versions.

We used Unix sorts and custom Java code to extract data-based features. In order to evaluate queries for producing ground truths and results features, we loaded the corresponding graphs into Virtuoso instances. We computed the cardinality estimation using the κ most common subjects and objects with $\kappa = 10$, $\kappa = 100$, and $\kappa = 1000$, selecting $\kappa = 1000$ as it offers better estimations when compared with the real cardinality of training queries.

Binary classification models. We experiment with well-known machine learning classifiers, including Decision Trees, Naive Bayes, Nearest Neighbours and Linear SVM. We also include a Random Baseline for comparison. We train and compare classifiers for different window sizes (specifically 3, 5, 9, corresponding to 2, 4 and 8 pairs of consecutive versions). For each query we evaluate whether or not the change in the query results can be predicted from the set of features extracted from the query itself and from the preceding dynamic graph. We split the data by query into 80% for training and 20% for tests, using 5-fold cross-validation to avoid overfitting.

Results. Tables 1 and 2 present, respectively, the results for OSC prediction on the Wikidata and DBpedia datasets. For reasons of space, we include only F_1 scores, where we refer to the accompanying online material for precision and recall [26]. We compare

[7] Truthy Wikidata versions provide values associated with the best non-deprecated rank for a given property (e.g., the most recent population), and thereafter omit qualifiers.

Table 1: F_1-score for OSC on the Wikidata dataset considering different features sets and window sizes (w)

	Classifier	Q	P	D	R	QPD	QPDR
w=3	*Random Baseline*	0.509	0.499	0.497	0.482	0.496	0.5
	Decision Trees	0.549	**0.554**	0.66	**0.855**	**0.61**	0.709
	Naive Bayes	0.522	0.36	0.478	0.825	0.498	0.67
	Nearest Neighbours	0.547	0.532	**0.691**	0.837	0.537	**0.83**
	Linear SVM	**0.582**	0.441	0.387	0.825	0.603	0.827
w=5	*Random Baseline*	0.487	0.482	0.495	0.501	0.515	0.504
	Decision Trees	0.539	**0.541**	**0.677**	**0.868**	**0.622**	0.735
	Naive Bayes	0.516	0.382	0.484	0.841	0.513	0.76
	Nearest Neighbours	0.54	0.536	0.656	0.837	0.524	**0.851**
	Linear SVM	**0.594**	0.442	0.472	0.841	0.612	0.838
w=9	*Random Baseline*	0.497	0.501	0.499	0.517	0.487	0.508
	Decision Trees	0.544	0.568	0.634	**0.876**	**0.631**	0.699
	Naive Bayes	0.53	0.464	0.498	0.859	0.528	0.781
	Nearest Neighbours	0.551	**0.568**	**0.675**	0.868	0.521	**0.863**
	Linear SVM	**0.595**	0.445	0.546	0.86	0.626	0.85

Table 2: F_1-score for OSC on the DBpedia dataset considering different feature sets and window sizes (w)

	Classifier	Q	P	D	R	QPD	QPDR
w=3	*Random Baseline*	0.503	0.514	0.494	0.498	0.486	0.491
	Decision Trees	**0.57**	0.488	**0.567**	**0.928**	**0.535**	0.888
	Naive Bayes	0.475	**0.532**	0.473	0.902	0.489	0.809
	Nearest Neighbours	0.519	0.474	0.507	**0.928**	0.481	0.904
	Linear SVM	0.474	0.474	0.504	**0.928**	0.509	**0.925**
w=5	*Random Baseline*	0.506	0.506	0.503	0.498	0.51	0.506
	Decision Trees	**0.571**	0.494	**0.583**	**0.938**	**0.581**	0.895
	Naive Bayes	0.475	**0.51**	0.525	0.936	0.536	0.858
	Nearest Neighbours	0.48	0.485	0.48	0.936	0.482	0.912
	Linear SVM	0.475	0.475	0.509	**0.938**	0.515	**0.937**
w=9	*Random Baseline*	0.491	0.508	0.522	0.499	0.493	0.5
	Decision Trees	**0.509**	0.502	0.505	**0.946**	0.521	0.883
	Naive Bayes	0.473	**0.507**	**0.525**	0.931	**0.536**	0.866
	Nearest Neighbours	0.474	0.503	0.47	0.942	0.481	0.914
	Linear SVM	0.475	0.475	0.516	0.94	0.516	**0.94**

the results for six feature sets: query (Q), property (P), degree-of-change (D), historical results (R), all features without historical results (QPD) and all features (QPDR). Given that different models perform better/worse in different settings, we highlight the best results in bold; Decision Trees had the best overall performance though it was outperformed by other models in multiple cases.

Comparing window sizes, we see that the best results overall are given for larger window sizes. This can be explained by the fact that variances of individual versions are smoothed out when longer intervals are considered. However, it is interesting to note that the performance improvement is not very pronounced, possibly because more recent changes are more similar to future changes.

Comparing different feature sets, we see that statistics based on historical query results (R) are the most important, and enable (by far) the most accurate predictions. In fact, the predictions with only results features are considerably better than the predictions combining all other features. Conversely, we find that query features and predicate features provide only slightly better predictions versus the random baseline. In the case of Wikidata, degree-of-change features offer notably better predictions than query or predicate-based features, especially when using the Nearest Neighbours or Decision Trees classifier. However, the quality of predictions drops for DBpedia, which we believe to be due to the relative sparsity of changes in the query results of the dataset, and also the more non-monotonic nature of changes vs. Wikidata.

6.2 Efficiency

Regarding online costs, historical results features were (by far) the most costly to compute. For the purposes of our experiments, we evaluated these queries sequentially over Virtuoso; each query added a cost of w times the query evaluation time on a single version based on evaluating the query on w independent copies for the w versions in the window. While queries on multiple versions could be parallelised, this would limit the parallel execution of individual queries. In future, RDF archiving techniques [9] could offer a way to optimise for querying multiple versions of an RDF graph. Conversely other features could be extracted near-instantaneously.

Summarising off-line costs, query features have no overhead as they require no indexes or statistics over the data. For dynamic predicates and degree of change, computing the graph summary (an $O(n \log n)$ process) for Wikidata was orders of magnitude slower than for DBpedia due to the scale (78 vs. 11 billion triples in total), but also the fact that Wikidata required computing deltas from complete versions, while DBpedia Live provided a single base version and its deltas as change sets (the graph summaries can be computed incrementally from deltas/changes). The greatest offline costs again involved the historical results, which required indexing the data in Virtuoso (requiring dictionary encoding and sorting in several orders for its indexes), and for building the training set, which required evaluating training queries over all non-withheld versions. Note that in the case of DBpedia we materialised each version for the purposes of evaluating the queries; again, a future option could be to rather leverage RDF archiving techniques.

7 CONCLUSIONS

In this paper, we have proposed a method for predicting whether or not the results of a query will change in the next version of a dynamic RDF graph. We propose four feature sets for this task, based on queries, predicates, degree-of-change, and historical results. For this task, we hypothesise that there exists a trade-off between the overhead of the framework and the accuracy of prediction: extracting richer data provides better features for prediction, but at the cost of extracting them. Our experiments confirm that features based on historical results provide by far the most accurate predictions. However, such features incur considerable overhead that may be unjustifiable in use-cases such as caching. In the case of Wikidata, degree-of-change estimates provide the next best alternative in terms of accuracy, which corresponds to our hypothesis; such features require a statistical summary rather than a complete index of historical versions. However, query features and dynamic predicates provide little improvement over a random baseline, while

degree-of-change estimates likewise show relatively poor accuracy over the daily DBpedia dataset where fewer changes are present. For future work, it would be of interest to explore further features that do not rely on historical results, to investigate the possibilities of reducing the overheads of computing historical results using RDF archiving techniques, and also to explore TTL predictions.

We refer to the online material for the datasets and queries used, along with additional results [26].

Acknowledgements. This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002, FONDECYT Grant No. 1181896 and CONICYT PFCHA/Doctorado Nacional/2017 - 21171070. We also thank the reviewers for their feedback.

REFERENCES

[1] Usman Akhtar, Muhammad Bilal Amin, and Sungyoung Lee. 2017. Evaluating scheduling strategies in LOD based application. In *Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 255–258.

[2] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *International Semantic Web Conference (ISWC)*, Vol. 8219. Springer, 277–293.

[3] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 330–341.

[4] Soheila Dehghanzadeh, Josiane Xavier Parreira, Marcel Karnstedt, Jürgen Umbrich, Manfred Hauswirth, and Stefan Decker. 2014. Optimizing SPARQL Query Processing on Dynamic and Static Data Based on Query Time/Freshness Requirements Using Materialization. In *Joint International Conference on Semantic Technology (JIST)*. Springer, 257–270.

[5] Renata Queiroz Dividino, Thomas Gottron, and Ansgar Scherp. 2015. Strategies for Efficiently Keeping Local Linked Open Data Caches Up-To-Date. In *International Semantic Web Conference (ISWC)*. Springer, 356–373.

[6] Renata Queiroz Dividino, Thomas Gottron, Ansgar Scherp, and Gerd Gröner. 2014. From Changes to Dynamics: Dynamics Analysis of Linked Open Data Sources. In *Dataset PROFIling & fEderated Search (PROFILES)*.

[7] Renata Queiroz Dividino, André Kramer, and Thomas Gottron. 2014. An Investigation of HTTP Header Information for Detecting Changes of Linked Open Data Sources. In *ESWC Satellite Events*. Springer, 199–203.

[8] Renata Queiroz Dividino, Ansgar Scherp, Gerd Gröner, and Thomas Grotton. 2013. Change-a-LOD: Does the Schema on the Linked Data Cloud Change or Not?. In *Consuming Linked Data (COLD)*. CEUR-WS.org.

[9] Javier D. Fernández, Axel Polleres, and Jürgen Umbrich. 2015. Towards Efficient Archiving of Dynamic Linked Open Data. In *DIACHRON Managing the Evolution and Preservation of the Data Web*. 34–49.

[10] Julien Genestoux, Brad Fitzpatrick, Brett Slatkin, and Martin Atkins. 2018. WebSub. W3C Recommendation. https://www.w3.org/TR/websub/.

[11] Larry González and Aidan Hogan. 2018. Modelling Dynamics in Semantic Web Knowledge Graphs with Formal Concept Analysis. In *World Wide Web Conference (WWW)*. ACM, 1175–1184.

[12] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[13] Aidan Hogan. 2015. Skolemising Blank Nodes while Preserving Isomorphism. In *World Wide Web Conference (WWW)*. ACM, 430–440.

[14] Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O'Byrne, and Aidan Hogan. 2013. Observing Linked Data Dynamics. In *ESWC*. Springer, 213–227.

[15] Kjetil Kjernsmo. 2015. A Survey of HTTP Caching Implementations on the Open Semantic Web. In *European Semantic Web Conference (ESWC)*. Springer, 286–301.

[16] Magnus Knuth, Olaf Hartig, and Harald Sack. 2016. Scheduling Refresh Queries for Keeping Results from a SPARQL Endpoint Up-to-Date (Short Paper). In *On the Move to Meaningful Internet Systems (OTM)*. Springer, 780–791.

[17] Magnus Knuth, Dinesh Reddy, Anastasia Dimou, Sahar Vahdati, and George Kastrinakis. 2015. Towards Linked Data Update Notifications Reviewing and Generalizing the SparqlPuSH Approach. In *Workshop on Negative or Inconclusive Results in Semantic Web (NoISE)*, Anastasia Dimou, Jacco van Ossenbruggen, Miel Vander Sande, and Maria-Esther Vidal (Eds.), Vol. 1435. CEUR-WS.org.

[18] Tomas Lampo, Maria-Esther Vidal, Juan Danilow, and Edna Ruckhaus. 2011. To Cache or Not To Cache: The Effects of Warming Cache in Complex SPARQL Queries. In *On the Move to Meaningful Internet Systems (OTM)*. Springer, 716–733.

[19] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[20] Johannes Lorey and Felix Naumann. 2013. Caching and Prefetching Strategies for SPARQL Queries. In *ESWC Satellite Events*. Springer, 46–65.

[21] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*. Springer, 376–394.

[22] Michael Martin, Jörg Unbehauen, and Sören Auer. 2010. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *Extended Semantic Web Conference (ESWC)*. Springer, 304–318.

[23] Paolo Missier, Pinar Alper, Óscar Corcho, Ian Dunlop, and Carole A. Goble. 2007. Requirements and Services for Metadata Management. *IEEE Internet Computing* 11, 5 (2007), 17–25.

[24] Mohamed Morsey, Jens Lehmann, Sören Auer, Claus Stadler, and Sebastian Hellmann. 2012. DBpedia and the live extraction of structured data from Wikipedia. *Program* 46, 2 (2012), 157–181.

[25] Alberto Moya Loustaunau and Aidan Hogan. 2019. Estimating the Dynamics of SPARQL Query Results Using Binary Classification. In *Querying and Benchmarking the Web of Data (QuWeDa)*. 5–20.

[26] Alberto Moya Loustaunau and Aidan Hogan. 2021. Online material. GitHub Repository. https://github.com/amoya87/sparqldynamics/.

[27] Sebastian Neumaier and Jürgen Umbrich. 2016. Measures for Assessing the Data Freshness in Open Data Portals. In *International Conference on Open and Big Data (OBD)*. IEEE Computer Society, 17–24.

[28] Chifumi Nishioka and Ansgar Scherp. 2016. Information-theoretic Analysis of Entity Dynamics on the Linked Open Data Cloud. In *Dataset PROFIling and fEderated Search for Linked Data (PROFILES)*. CEUR-WS.org.

[29] Chifumi Nishioka and Ansgar Scherp. 2017. Keeping Linked Open Data caches up-to-date by predicting the life-time of RDF triples. In *International Conference on Web Intelligence (WI)*. ACM, 73–80.

[30] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2015. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *SIGMOD International Conference on Management of Data*. ACM, 1777–1792.

[31] Alexandre Passant and Pablo N. Mendes. 2010. sparqlPuSH: Proactive Notification of Data Updates in RDF Stores Using PubSubHubbub. In *Scripting and Development for the Semantic Web*. CEUR-WS.org.

[32] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[33] Qun Ren, Margaret H. Dunham, and Vijay Kumar. 2003. Semantic Caching and Query Processing. *IEEE TKDE* 15, 1 (2003), 192–210.

[34] Mariano Rico, Rizkallah Touma, Anna Queralt, and María S. Pérez. 2018. Machine Learning-based Query Augmentation for SPARQL Endpoints. In *Web Information Systems and Technologies (WEBIST)*. SciTePress, 57–67.

[35] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *International Semantic Web Conference (ISWC)*. Springer, 261–269.

[36] Heiner Stuckenschmidt. 2004. Similarity-Based Query Caching. In *International Conference on Flexible Query Answering Systems (FQAS)*. Springer, 295–306.

[37] Sebastian Tramp, Philipp Frischmuth, Timofey Ermilov, and Sören Auer. 2010. Weaving a Social Data Web with Semantic Pingback. In *Knowledge Engineering and Management (EKAW)*. Springer, 135–149.

[38] Jürgen Umbrich, Michael Hausenblas, Aidan Hogan, Axel Polleres, and Stefan Decker. 2010. Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In *Linked Data on the Web (LDOW)*. CEUR-WS.org.

[39] Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira. 2012. Hybrid SPARQL Queries: Fresh vs. Fast Results. In *International Semantic Web Conference (ISWC)*. Springer, 608–624.

[40] Jürgen Umbrich, Marcel Karnstedt, and Sebastian Land. 2010. Towards Understanding the Changing Web: Mining the Dynamics of Linked-Data Sources and Entities. In *Lernen, Wissen & Adaptivitat (LWA)*. 159–162.

[41] Jürgen Umbrich, Nina Mrzelj, and Axel Polleres. 2015. Towards capturing and preserving changes on the Web of Data. In *Managing the Evolution and Preservation of the Data Web (DIACHRON)*. 50–65.

[42] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.

[43] Gregory Todd Williams and Jesse Weaver. 2011. Enabling Fine-Grained HTTP Caching of SPARQL Query Results. In *International Semantic Web Conference (ISWC)*. Springer, 762–777.

[44] Gang Wu and Mengdong Yang. 2012. Improving SPARQL query performance with algebraic expression tree based caching and entity caching. *J. Zhejiang Univ. Sci. C* 13, 4 (2012), 281–294.

[45] Wei Emma Zhang, Quan Z. Sheng, Kerry Taylor, and Yongrui Qin. 2015. Identifying and Caching Hot Triples for Efficient RDF Query Processing. In *Database Systems for Advanced Applications (DASFAA)*. Springer, 259–274.