Hybrid SPARQL queries: fresh vs. fast results

Hybrid SPARQL queries: fresh vs. fast results

Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira

Digital Enterprise Research Institute, National University of Ireland, Galway
firstname.lastname@deri.org

Abstract. For Linked Data query engines, there are inherent trade-offs between centralised approaches that can efficiently answer queries over data cached from parts of the Web, and live decentralised approaches that can provide fresher results over the entire Web at the cost of slower response times. Herein, we propose a *hybrid query execution* approach that returns fresher results from a broader range of sources vs. the centralised scenario, while speeding up results vs. the live scenario. We first compare results from two public SPARQL stores against current versions of the Linked Data sources they cache; results are often missing or out-of-date. We thus propose using *coherence estimates* to split a query into a sub-query for which the cached data have good fresh coverage, and a sub-query that should instead be run live. Finally, we evaluate different hybrid query plans and split positions in a real-world setup. Our results show that hybrid query execution can improve freshness vs. fully cached results while reducing the time taken vs. fully live execution.

1 Introduction

As of today, there are an estimated 30 billion facts published on the Web as Linked Data [3]. Traditional approaches for querying Linked Data rely on optimised, centralised RDF stores that *cache* remote content [2,17,4]. However, maintaining comprehensive and up-to-date cached data is an impossible task. First, the coverage of centralised stores is limited by the amount of data that can be located, retrieved and indexed by local servers. Second, result freshness is determined by the last time the relevant Web documents were cached, which can often be measured in days or even months, leading to stale query answers.

Conversely, various authors have proposed techniques to process queries live and directly over Linked Data [6,20,8,10], bypassing the need for maintaining a replicated (full) store. In live-querying scenarios, coverage spans the Web, and the freshness of results depends on live-query latency, which can generally be measured in terms of minutes or even seconds. However, in live approaches, retrieving remote content from diverse sources at query-time naturally implies much slower response times compared to querying centralised stores.

Thus, as shown in Figure 1, there is an inherent trade-off between query approaches that give fresh results versus approaches that give fast results, represented at two ends by centralised SPARQL stores and live-query techniques respectively. Aside from performance, the *recall* of answers is also crucial to consider. Figure 2 shows how results from a live query approach (L) and results

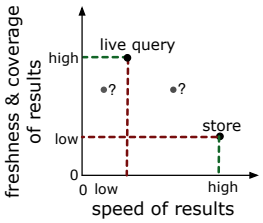


Fig. 1. Query Trade-off

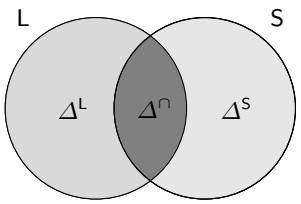


Fig. 2. Result set (Venn) diagram

from a store of cached Web data (S) may diverge. Some of the results remain the same ($\Delta^{\cap}$); when this happens we say that the SPARQL store returned *coherent* results. However, the store may return results that live querying does not (Δ^{S}), which may be stale results from sources that have changed, or may be accurate results from sources that live querying did not consider. Conversely, the live approach may find answers that the store did not (Δ^{L}), either from updates in remote sources, or from sources not cached/indexed by the store.

Evidently, both centralised and live query engines have their inherent strengths and weaknesses. We thus propose a hybrid approach that combines the two. Our hybrid engine can be thought of as a "live-wrapper" for centralised SPARQL endpoints that splits a query into two, where one part is executed over the centralised store and the other part is executed using existing live-querying techniques. By getting the store to quickly service query-patterns for which it has good up-to-date coverage, and by running the rest of the query live, our hybrid approach aims to strike a balance between fresh and fast results.

Deciding which parts of the query to run live and which to run centrally requires knowledge of the *coherence* of the replicated store wrt. remote data, which depends on the dynamicity of remote data and coverage of the store. Consider the hypothetical query What are the current temperatures in European capital cities? Data about current temperatures are dynamic and appropriate for live-query techniques (the store would return stale results). Also, the store may not have indexed data about which continent each city is on; this part should also be fetched live. Conversely, information about capital cities is static and well-covered by the store. In our hybrid approach, up-to-date results about capital cities are quickly retrieved from the store and enriched with results from the Web about continents and temperatures. Stale results are avoided by not asking the centralised store about current temperatures.

We continue this paper with background on Linked Data querying (Section 2). We then introduce our hybrid query architecture (Section 3). Next, we propose probing RDF stores to collect coherence estimates, and present experiments for two public SPARQL engines (Section 4). We then detail our hybrid query-planning component (Section 5). For the two public stores, we evaluate different hybrid query plans and the extent to which they speed up live querying while freshening up centralised results (Section 6). We then conclude (Section 7).

2 Background

Centralised Linked Data stores execute SPARQL queries over a local copy of Web documents. Some endpoints, like FactForge [2], index selected subsets of Linked Data. Other systems, like OpenLink's LOD cache[1] and the Sindice [17] SPARQL endpoint[2] (both powered by Virtuoso [4]) aim to have broad coverage of Linked Data. However, as we show in Section 4, constantly maintaining a broad, fresh coverage of remote data is unfeasible in such setups.

Recently, various authors have proposed methods for performing live querying, accessing remote data *in situ* and at runtime. Ladwig and Tran [8] categorise these approaches as: (i) top-down, (ii) bottom-up, and (iii) mixed strategy. Top-down evaluation determines remote, query-relevant sources using a *source-selection index*: a local repository summarising information about sources that can vary from inverted-index structures [17,10], to query-routing indexes [16], schema-level indexes [15], or hash-based summaries [20]. The bottom-up query evaluation strategy discovers relevant sources on-the-fly during the evaluation of queries by selectively and recursively following links starting from a "seed set" of URIs taken from the query [6]. The third strategy uses (in a top-down fashion) some knowledge about sources to generate the seed list, then discovering additional relevant sources using a bottom-up approach [8]. These three approaches rely on time-consuming remote lookups, but conversely offer fresh results.

In this paper, we use the bottom-up approach proposed by Hartig et al. [6], called link-traversal based query execution (LTBQE), to provide live results. LTBQE uses dereferenceable URIs in a query to find remote documents relevant to that query. During query execution, dereferenceable links that match query patterns are followed to discover further relevant data. We choose LTBQE for live querying as it requires no local knowledge and thus, in the hybrid scenario, can find sources that the store may not even be aware of. An inherent weakness of LTBQE is its dependence on the availability of dereferenceable URIs. However, as the uptake of Linked Data principles continues, we expect the ratio of dereferenceable data on the Web to increase. We refer the reader to our previous work which analyses the prevalence of dereferenceable URIs and the ratio of information about RDF resources that is dereferenceable on the Web [19].

While the above approaches access data directly, an orthogonal approach to live querying is that of federated SPARQL, where queries are executed over a group of possibly remote endpoints [12,13,1]. Given the recent proliferation of SPARQL endpoints on the Web of Data [3], federation is a timely topic that enjoys increasing attention [5]. Our approach could also use live federated techniques, though we would need to ascertain the freshness of remote endpoints.

Like us, various authors have discussed the combination of local (central) and remote (live) querying techniques on a theoretical, optimisation, engineering and social level (e.g., [9,22]). However, to the best of our knowledge, no-one has tackled the question of deciding which parts of a query are suitable for local/remote execution; here, we propose making such a split based on dynamicity estimates.

[1] http://lod.openlinksw.com/sparql
[2] http://sparql.sindice.com/

In this light, our approach relates to the broad field of research on (Web) caching and the problem of guaranteeing cache coherence [11], as well as semantic caching in, e.g., mediator systems [7]. However, instead of splitting queries, such systems often apply an all-or-nothing approach, either relying solely on locally cached data or going entirely live. More recently, various authors have discussed invalidation of SPARQL caches [23] but rely on monitoring inserts within the local system, and are not concerned with the dynamics of remote data.

The work in hand is based on the idea of hybrid SPARQL queries originally proposed by us in [22]. In [21], we discussed a wide range of general strategies to implement the proposed hybrid query planning and processing approach. The current paper is based on the conclusions drawn therein and proposes a first concrete instantiation of an according query engine including a comprehensive experimental evaluation over real datasets.

3 Architecture of a Hybrid Query Engine

Our proposed hybrid query engine has the following targets: (T1) **fast response times** close to those of centralised queries; (T2) **coherence of results** close to those of live query processing such that we retrieve fresh answers; (T3) **system independence**, i.e., being compatible with any SPARQL-enabled store or live-query processor; and (T4) **lightweight implementation** with low resource requirements, particularly regarding main memory.

As previously discussed, (T1) and (T2) are antagonist targets: thus, the main component in the architecture is the QUERY PLANNER which tries to find an overall "optimal" trade-off for a given request, deciding what parts of the query to delegate to the live engine and to the store. With regards to (T3), we can initialise our architecture with an INDEX QUERY INTERFACE and a LIVE QUERY INTERFACE as black-box components; both consume SPARQL queries and produce SPARQL results, but the former interfaces with a central store, whereas the latter interfaces with the Web. Finally, to help find that trade-off, the COHERENCE MONITOR collects *high-level* empirical statistics (see Section 4) about the store's coverage of data for different query patterns compared with the Web. These compact estimates have (relatively) low maintenance costs (as per (T4)). The resulting architecture is illustrated in Figure 3.

The INDEX QUERY INTERFACE can be a (possibly remote) public SPARQL store or any data warehousing approach that offers the SPARQL protocol (e.g., an intranet database). The LIVE QUERY INTERFACE also accepts SPARQL queries and could be based on, for instance, a bottom-up link-traversal engine [6] or a top-down source selection index [20], or some combination thereof. Here we instantiate the live query processor with a bottom-up, link-traversal based query execution approach (LTBQE) as originally proposed by Hartig et al. [6].

The COHERENCE MONITOR collects information about the coverage and freshness of different triple patterns and sources. The coherence estimates of individual patterns is used by the QUERY PLANNER component to split a given query into two sub-queries—a central and a live sub-query. Eventually, the query processor forwards the central part to the index query interface and the live part

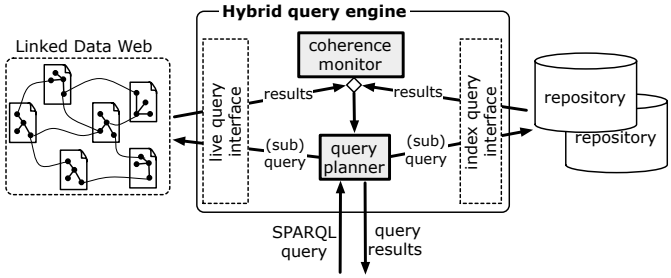


Fig. 3. Architecture of a hybrid query-engine

is processed over the relevant Web sources *in situ*. We see this conceptually straightforward architecture as a first step towards freshening up centralised results: topics such as adaptive coherence estimates and more fine-grained interaction between the central and remote query processors are left to future work.

Note that since the central SPARQL store is treated as a black-box (as per (T3)), we cannot influence the design of the physical plan for the static sub-query: we delegate generating the final sub-query plan to the engine, which we assume implements, e.g., local selectivity estimates to organise optimal execution. In the general case, a similar situation exists for the live query processor.

In the following sections, we elaborate further on the COHERENCE MONITOR component (Section 4) and the QUERY PLANNER component (Section 5).

4 Coherence Estimation

Given the scope [3] and dynamicity [18] of Linked Data, results returned by a centralised endpoint are inherently limited by its coverage of the Web and by the freshness of its local index. The COHERENCE MONITOR computes and stores the coherence estimates of query patterns for a centralised endpoint in contrast to the fresh results given by live query execution.

For a given endpoint, we issue the same set of queries against both the store and the live engine and compare the results, identifying data predicate–source combinations that are likely to be stale. For testing, we chose the two aforementioned SPARQL stores covering a broad range of Linked Data: "the Semantic Web Index" hosted by Sindice, and the "LOD Cache" hosted by OpenLink. We randomly sampled 12,000 URIs from the 2011 Billion Triple Challenge dataset[3], which covers around 8 million RDF Web documents.[4] For each URI, we generated the following query.

[3] http://challenge.semanticweb.org/

[4] We considered using the SPARQL 1.1 `SAMPLE` keyword, but Virtuoso does not support SPARQL 1.1 (though it does support similar custom syntax). Further, `SAMPLE` makes no guarantees about the randomness of results.

```
SELECT ?sIn, ?pIn, ?oOut, ?pOut
WHERE { ?sIn ?pIn <entityURI> . <entityURI> ?pOut ?oOut . }
```

This query returns all values of RDF triples in which the given entity URI appears in either the subject or object position. We then compare the set of store results (S) to the set of live results (L). We view results as consisting of sets of sets of variable bindings (i.e., $\mathsf{S}, \mathsf{L}, \Delta^* \subset 2^{\mathbf{V}\times\mathbf{UL}}$ reusing common notation for the set of all query variables, URIs and literals resp.); we exclude answers that involve blank nodes to avoid issues of scoping and inconsistent labelling.

We reuse notation outlined in Figure 2. $\Delta^{\cap} := \mathsf{L} \cap \mathsf{S}$ refers to the set of results in both L and S, i.e., results for which the store is up-to-date. $\Delta^{\mathsf{L}} := \mathsf{L} \setminus \mathsf{S}$ refers to the set of results returned by the Web that are not returned by the store. $\Delta^{\mathsf{S}} := \mathsf{S} \setminus \mathsf{L}$ indicates results returned only by the store. We add subscripts to indicate results for a certain query, e.g., Δ^{L}_q. We denote results for a predicate p as, e.g., $\Delta^{\mathsf{L}}_q(p) := \{r \in \Delta^{\mathsf{L}}_q : (\texttt{?pIn}, p) \in r \vee (\texttt{?pOut}, p) \in r\}$, and say $p \in \Delta^{\mathsf{L}}_q$ iff $\Delta^{\mathsf{L}}_q(p) \neq \emptyset$.

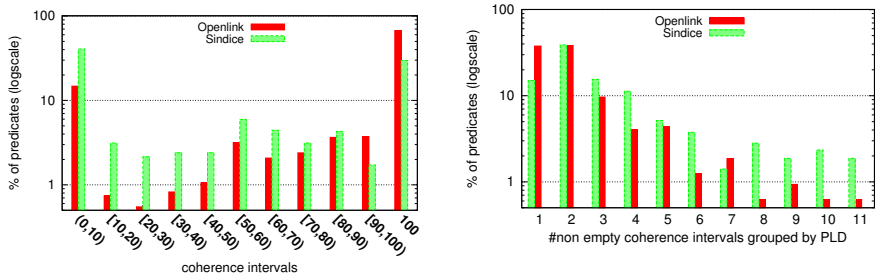
To ensure lightweight statistics with broad applicability, our notion of coherence for query patterns centres around predicates. This restricts our approach to triple patterns with a constant in the predicate position; other patterns are assigned a default estimate. We ran our experiments in early March 2012 and gathered coherence information for 2,550 predicates for OpenLink and 1,627 predicates for Sindice.[5] To quantify the coherence of individual predicates based on the results, we define the *result-based coherence* measure, which computes the ratio of missing results for a predicate p. For the full set of queries $\mathcal{Q}$, let $M_r(p)$ denote the count of all live results involving the predicate p that were missed by the store, summated across all queries ($M_r(p) = \sum_{q\in\mathcal{Q}} |\Delta^{\mathsf{L}}_q(p)|$). Let $L_r(p)$ denote the count of all results involving p retrieved from the live engine ($L_r(p) = \sum_{q\in\mathcal{Q}} |\mathsf{L}_q(p)|$). Result-based coherence is then:

$$\mathsf{coh}_r(p) = 1 - \frac{M_r(p)}{L_r(p)} \quad .$$

There are other alternatives to measure coherence for a store [21]. Since we could not empirically observe significant differences among the different measures discussed in [21], we use the result-based coherence measure in this paper.

For the two stores under analysis, Figure 4(a) illustrates the number of predicates that fall into different intervals of coherence values; the y-axis is in logarithmic scale, and the linear x-axis intervals represent the coherence measures as percentages (the right of the graph indicates increasingly coherent predicates). The figure shows that the OpenLink endpoint is more in sync with current Web data than Sindice; we believe that OpenLink was extensively updated in Feb. 2012. We measured that 67% of the tested predicates in the OpenLink index are entirely up-to-date ($\mathsf{coh}_r(p) = 1$), versus 30% of the predicates for the

[5] The statistics took over a week to collect politely (5 s delay). Maintaining coherence statistics is non-trivial, but out of scope. Discussion can rather be found in [21, § 4].



(a) Coherence distribution (b) PLD coherence variations

Fig. 4. Distribution of predicate coherence values and variation across PLDs

Sindice endpoint. In contrast, information for 14% of the tested predicates in the OpenLink index are entirely missing or out-of-date ($\mathsf{coh}_r(p) = 0$), versus 40% for Sindice; these high percentages are due to partial coverage of Web sources, outdated data-dumps in the index, and predicates with dynamic values.

In more detail, Table 1 shows the top 5 predicates where $\mathsf{coh}_r(p) = 0$ for both stores, ordered by the number of queries in which they featured as a result. First, we see a mix of dynamic time-stamp predicates that change for every access or modification of a document (`swivt:creationDate`, `swivt:wikiPageModificationDate` and `aims:hasDateCreated`). Second, we see predicates not covered by the index. For Sindice, the incoherent `*:doi` predicates are due to a lack of coverage of the `dx.doi.org` domain and the high incoherency of `skos:` predicates is due to bulk changes in the `esd-toolkit.eu`, `esd.org.uk` and `bio2rdf.org` domains; for OpenLink, the incoherency of `vitro:mostSpecificType` relates to data on the `cornell.edu` domain.

We further analysed the correlation for coherence estimates of the same predicates *across* the two stores. We used Kendall's τ, which measures the agreement in ordering for two measures in a range of $[-1, 1]$, where -1 indicates perfectly inverted ordering and 1 indicates the exact same ordering. The τ-score across the two stores was 0.16, with a negligible p-value, indicating a weak, significant and positive correlation between the coherence of predicates for the two stores. The low correlation highlights the store-specific nature of these measures, which are as much about index coverage than about the dynamicity of values. As such, our approach tackles both the global problem of dynamicity and the central problem of store coverage.

Finally, we looked at the correlation between the selectivity of predicates (i.e., how often they occur) and their coherence, which may have potential consequences for query planning. Specifically, for each store, we compared the number of (Web) results generated for each predicate across all queries and their $\mathsf{coh}_r(p)$ value. The τ-value for OpenLink was 0.1, indicating that less selective patterns tend to have slightly lower coherence; the analogous τ-value for Sindice was -0.03, indicating a *very* slight correlation in the opposite direction. Though

Table 1. Most dynamic and prevalent predicates

№	OpenLink		Sindice	
	pred.	*queries*	*pred.*	*queries*
1	`swivt:creationDate`	510	`swivt:creationDate`	118
2	`vitro:mostSpecificType`	104	`skos:narrower`	48
3	`swivt:wikiPageModificationDate`	45	`skos:historyNote`	43
4	`aims:hasDateCreated`	42	`bibo:doi`	34
5	`madsrdf:hasCloseExternalAuthority`	31	`prism21:doi`	34

limited, we take this as anecdotal evidence to suggest that correlation between the selectivity and coherence of predicates is weak, if any.

Above, we naïvely assume a single coherence value for predicates in all cases, ignoring subject or object URIs: keeping information for each subject/object would have a high overhead. However, we can generalise subject/object values into pay-level-domains (PLD)[6] and then track coherence for predicate–domain pairs. Thus, we mapped the entity URIs of the queries to their PLDs (581 PLDs with a maximum of 74 queries per domain) and resolved the coherence of predicates for individual PLDs. Focussing on the coherence measure, we divided the scores into eleven intervals as per the x-axis of Figure 4(b), and for each predicate, count how many intervals it falls into for different PLDs. We observe that the subject and object URIs can be ignored for roughly 40% of the OpenLink and roughly 15% for the Sindice predicates. However, we see the importance of tracking coherence for predicate–domain pairs for the remaining predicates. The plurality of predicates (∼40%) show two intervals of coherence values.

5 Query Planner

Our hybrid engine combines centralised and decentralised/live query execution to obtain a balance between fresh and fast results. The QUERY PLANNER is responsible for splitting the query into a part for execution against the centralised store and a part for live execution. We focus on evaluating conjunctive queries (i.e., SPARQL BGPs). Other SPARQL features—except `OPTIONAL` (and `MINUS` & [`NOT`] `EXISTS` in SPARQL 1.1) for which LTBQE is ill-suited since it (typically) does not operate over a pre-defined dataset—can be layered on top.

As indicated before, we argue for a single split of the query plan into one "central" (executed by the store) and one live part. While in theory it would also be possible to use multiple splits, the resulting intertwined dependencies between the central and live parts would lead to very complex query planning, and would require shipping bindings back and forth between the two engines. Thus, advanced splitting approaches are better suited to controlled environments (i.e., not public endpoints). We discuss this issue in more detail in [21].

Given the focus on a split into (at most) two parts, the results of the first part serve as input bindings for the second part. We must then decide whether

[6] A pay-level-domain is the domain name one has to register and pay for.

the central part is processed first (i.e., at the bottom of the query plan) or last (i.e., at the top of the plan). As previously discussed in [21], there are constraints inherent to live SPARQL query execution methods, such as the need for dereferenceable URIs in some of the query patterns. By running the central part first, we do not only obtain the store's results more quickly, but also provide additional dereferenceable URI bindings for the live querying phase (passed to the live engine using the `VALUES` [previously `BINDINGS`] clause in SPARQL 1.1).

As per traditional database query-planning approaches, we must then decide the execution order of (commutative) join patterns. However, instead of only optimising for speed, we now *also* wish to optimise for freshness. An intuitive approach, which we call **coherence-based ordering**, is to build a query plan with the most coherent patterns at the bottom for central execution, and the most incoherent patterns at the top for live execution. This increases the likelihood that the final result set is fresh and it limits the number of patterns executed live. However, the most coherent patterns may also be the least *selective* (i.e., return the most bindings) thus inflating the number of intermediate results to process. Consequently, this can increase the number of bindings for the patterns executed live, potentially hurting the performance. Because of this and backed by the absence of correlation between coherence and selectivity, we also consider another approach following traditional **selectivity-based reordering**, where the most restrictive patterns are executed first reducing intermediate results.

After selecting an ordering, we must also select a *split pattern*. The split pattern is the position in the query plan in which the query is divided into the two parts. Everything below the split is executed centrally, and everything above and including the split pattern is executed live. Following the same intuition of executing low-coherence patterns live, one option is to choose the **most incoherent** triple pattern as the split. However, the central store may still receive highly incoherent patterns (below the max) for which it will return incoherent results. Another approach is to define a constant value indicating a **threshold of incoherence**, where the lowest pattern breaching the threshold becomes the split pattern; this ensures that the store does not receive patterns that are highly incoherent. A further option is to split by a **fixed position** n, whereby the n bottom patterns are executed by the store and the rest are run live. Choosing between the different split options affects the core trade-off of fresh vs. fast results, and thus could be parameterised for individual user needs.

Figure 5 depicts examples of hybrid query plans, including our two different choices of ordering, as well as some possible split choices. In the selectivity-based ordering, we see that different types of coherence thresholds may lead to more patterns being run live than when explicitly ordered by coherence. Conversely, at the base of the plan for the coherence-based ordering, we see that **tp4** will return a lot of intermediate bindings (since it has a low selectivity) and does not share a join variable with **tp1** on the right-hand side of the join. In general, one may expect that the selectivity-based operator order would provide low answer times by minimising intermediate bindings, but would return less fresh results since low coherence patterns can appear below the split. However, this ordering also ends up pushing more patterns live since patterns with low selectivity and high

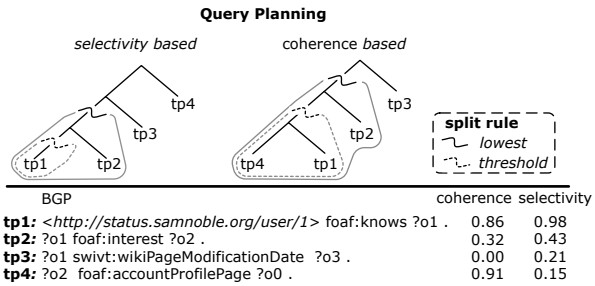


Fig. 5. Example hybrid query plans for different orderings and splits

coherence are often above the split. Conversely, a coherence-based ordering will lead to more intermediate results, but will run more patterns centrally. Thus, a general conclusion about which ordering is preferable is not possible; we instead compare combinations of orderings and splits on an empirical basis in Section 6.

In practice, for selectivity ordering, we create our hybrid SPARQL query plan using ARQ based on a "variable counting" technique [14], which estimates the selectivity of different triple patterns based on rules involving the number and position of variables it contains. This could be replaced with cost-based planning using empirical selectivity estimates, but we would need statistics about the underlying data. Thus, following a rule-based approach is more in line with targets (T3) system independence, and (T4) lightweight implementation (cf. Section 3). A coherence-based operator order is supported by reordering the triple patterns in the query plan produced by ARQ based on their coherence values.

In fact, since we consider the store and the live-query component as black boxes, both sub-queries will be reordered by the respective engines, thus mitigating some of the performance penalty associated with the possibly naïve ordering used to decide the split in the hybrid query plan. For example, referring back to the coherence-ordered plan of Figure 5, if the lowest coherence split rule is applied, the store may internally decide to run **tp1**, **tp2** and then **tp4** in that order, avoiding the (huge) expense of running **tp4** first.

6 Evaluation

We now evaluate our proposed hybrid query execution. Our concrete goals are as follows: (i) to prove concept in a realistic setting and show that with the correct plan, hybrid query execution can extend and freshen up central results while speeding up live results; (ii) to evaluate different query plan strategies by comparing (ii_a) selectivity- and coherence-based ordering and (ii_b) different split strategies for the query planning. In parallel, we are interested to see how useful our coherence estimates are for the hybrid-query planning phase.

To do so, our evaluation is run against the two selected public endpoints: Sindice and OpenLink. For this, we require a set of evaluation queries that are answerable by a Linked Data query engine. We would like these queries to

have broad coverage of diverse Web sources in order to properly test coherence estimates and hybrid splits. Hence, we generate queries from the Billion Triple Challenge 2011 dataset, which covers a broad range of Web documents. We apply a random walk technique on the dataset, selecting random paths between dereferenceable URIs in the data to produce queries that will give non-empty results if executed with the LTBQE LIVE QUERY INTERFACE (see [19] for more details). Using this method, we produce 200 SPARQL `SELECT` queries of different shapes (star, path, mixed), with varying numbers of patterns (2–6), randomly assigned distinguished variables, and at least one pattern above and below a coherence threshold of 0.5 (i.e., suitable for hybrid execution). After filtering out queries that produce empty results (e.g., due to offline sources) or result in endpoint errors (like timeouts), we obtained a set of 98 stable queries for the OpenLink store and 91 stable queries for the Sindice store. We reran all experiments four times over a period of eight days to verify repeatability.

To evaluate different orders and different cut-off positions, we created hybrid query plans for each query using both the selectivity- and coherence-based reordering strategies. Each query plan is then run entirely live, entirely by the store, and also run for every possible split position in both orders where part goes live and part goes to the store. We execute all split positions for simple convenience: we can then later compare different *a priori* strategies for picking a single split by looking up the corresponding results (and not rerunning queries).

In terms of the repeatability of results, for each configuration, we measured deviations for recall of results and query time across the four runs vs. the best approach (highest recall, lowest time). We then calculated the mean of these absolute deviations across all queries. For the live (LTBQE) execution, we measured a recall deviation of 3% and a time deviation of 2.7%. For the various other configurations, the recall deviation varied between 0–5% for OpenLink and between 0–2% for Sindice. Although the recall of the stores was very stable, we observed average time deviations of up to 36% for OpenLink and 17% for Sindice, indicating variable query response times. Acknowledging that public endpoints and remote data sources can be unstable, we wish to factor out this instability to derive comparable results across different hybrid strategies (we wish to compare different hybrid query plans, not the performance of public SPARQL endpoints). Thus, to avoid outliers, for each query and each configuration, we only select the best run in terms of recall, and in case two runs have the same recall, we select the one with the lower query time.

We first focus on ordering. To initially prove concept, we want to show that, in practice, hybrid query execution can *potentially* improve the freshness of results vs. the store while reducing query time vs. live querying. Table 2 presents such an analysis for both stores, where we see the potential percentage of queries that can be improved using our hybrid approach for both orders assuming (for the moment) that the best possible split position is picked (i.e., given the results, we select the split position that gave the highest recall and if tied, the lowest time; we evaluate split-selection strategies later). Recall is measured relative to the live results, which we know to be fresh. For OpenLink, we see that the recall of the store can only be improved for roughly half of the queries; however, the recall

Table 2. For both stores, the percentage of queries that can potentially be improved for each order assuming the best split position is picked

improvement	OpenLink		Sindice	
	`sel`	`coh`	`sel`	`coh`
Better Than Centralised Recall:	43%	53%	87%	91%
Better Than or Equal Centralised Recall:	97%	100%	99%	97%
Better Than Live Time:	92%	45%	16%	3%
Better Than Centralised Recall & Live Time	39%	13%	8%	1%
Better Than Or Equal Centralised Recall & Live Time:	92%	45%	16%	3%

of the store is already 1 in many cases and cannot be improved, only equalled. Equal ties in time are much more rare. In terms of improving the time for live results, the `sel` ordering seems much more beneficial for OpenLink than `coh`, likely due to fewer intermediate results being generated in the former ordering: `sel` improves or equals the store's recall while improving the live time in 92% of the queries. For Sindice, we found that the store often returned no query results: 84% of the queries ran entirely live as a fallback. Thus, the recall of many queries can be improved outright, but few queries are faster than the live approach. Since only 16% of the queries for Sindice are run in a truly hybrid fashion, we henceforth focus on OpenLink—all hybrid query results for Sindice were very close to the live approach. Table 2 shows that, in an ideal case, the hybrid approach can indeed improve result freshness while reducing the time required to process queries. Furthermore, the chosen ordering strategy seems to have a clear impact on freshness and query time.

We now compare concrete split strategies that *a priori* select a split based on the query and coherence/selectivity estimates (i.e., as would be required for hybrid query planning in reality, where an ordering and a split strategy are sufficient to generate a plan). We also look at the *degree* to which central recall is improved and live querying is sped up. To compare different splits and ordering combinations, we first filter out queries that, across the four runs, did not provide results for all possible split positions and orderings for one or more of the setups. We also removed queries with only two patterns, for which the choice of split is trivial. This results in a final set of 43 queries.[7]

For each ordering and for a variety of different split strategies, Figure 6 plots the aggregate speed up and recall ratio versus live querying. Specifically, to calculate speed up, the total time taken by the live approach to run all queries is divided by the total time taken for each individual approach; e.g., a speed up of 6 indicates that the approach in question was 6× faster than live querying. Conversely, recall is measured by taking live querying as the gold standard. Live querying is thus placed at $(1, 1)$. Orderings are intuitively represented by `sel-*` and `coh-*`. The best split approaches are represented by `*-best`; these splits cannot be determined before query execution, but rather represent the ideal case. Splitting at the most incoherent pattern is indicated by `*-incoh`. Using a coherence threshold of 0.5 to perform the split is indicated by `*-thr`. A random

[7] Queries are available online at the following address: http://code.google.com/p/lidaq/source/browse/queries/iswc-2012.tar.gz

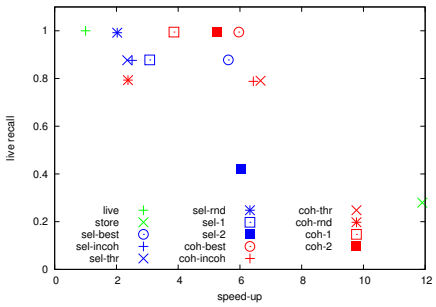


Fig. 6. Recall vs. speed-up trade-off for different hybrid plans (`thr` $= 0.5$)

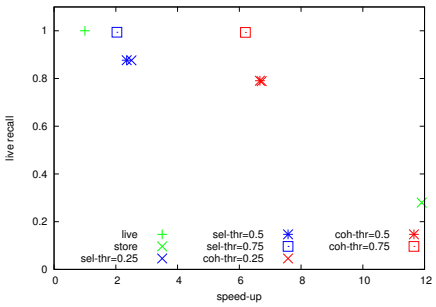


Fig. 7. Recall vs. speed-up trade-off for different thresholds

split (i.e., a guess) is indicated by `*-rnd`. Fixed split positions are indicated by `*-1` and `*-2` for $n = 1, 2$. Note that the threshold strategies `*-incoh`/`*-thr` can go fully live or fully central depending on the coherence values found for the query, whereas `*-best`, `*-rnd`, `*-1` and `*-2` must split the query. Figure 7 shows the same analysis, but for varying coherence threshold values.

In fact, both plots offer an empirical version of the trade-off introduced in Figure 1, where our hybrid strategies sit between live querying and the store. For both graphs, we see that the store is the fastest, and about $12\times$ faster than the live approach; however, recall is poor. Perhaps the best hybrid approach is `coh-thr=0.75` in Figure 7, which maintains an almost perfect recall but offers a speed up of more than $6\times$ live querying, slightly beating the ideal of `coh-best` (which must split the query). Interestingly, some fixed-position split strategies—particularly `coh-2` in Figure 6, which is $\sim 5\times$ faster than live, but maintains an almost perfect recall—can approach the ideal of `coh-best` quite closely.

While such an aggregated view presents high-level insights into the overall performance of the different strategies, we cannot identify the distribution over the queries in terms of achieved freshness and time. This is supported by Figure 8 and Figure 9, which show the recall for each query and the query time for each query respectively. We plot the number of queries on the x-axis that achieve a certain recall or time ratio shown on the y-axis. All 43 queries are sorted for each approach separately, providing a global view on each performance, but not supporting a per-query comparison of the strategies. While querying the store results in the fewest queries with a recall of 1, it also results in the most queries with a recall between 0 and 1. On the contrary, `coh-best` and `coh-2` are tied for keeping 100% recall across 42 queries and provide 0% only for the last query. Interestingly, most queries run with any hybrid strategy result in a recall of either 1 or 0. As expected, Figure 9 shows that query times for the centralised store are far below all other approaches in most cases. However, it is in fact slower for 2 anomalous queries that OpenLink struggles with.[8] Though all hybrid strategies were as fast or faster than live querying ($y = 1$) in all cases,

[8] One such example at the time of writing was `http://bit.ly/IPRec9`.

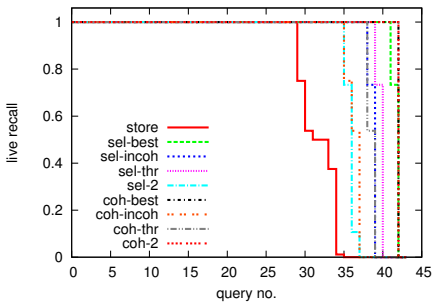


Fig. 8. Queries ordered by *recall* for different order and split strategies

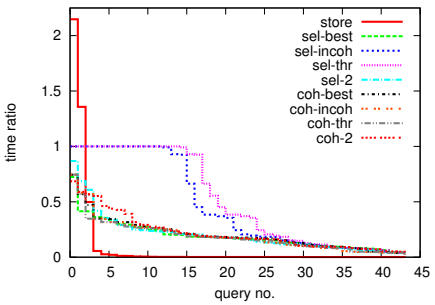


Fig. 9. Queries ordered by *time* for different order and split strategies

we see that `sel-incoh` and `sel-thres` did not show a time improvement for around 13 queries, where they were likely run completely live.

Eventually, we are interested in how recall and query times compare for different approaches on the same queries. Thus, in Figure 10 and Figure 11 we ordered the queries for each approach identically (using the results of the store for ordering). This means that in these figures each point on the x-axis presents the *same* query for each approach. In this case, plotting the absolute values would not allow any meaningful insights due to the ups and downs that each plot would show. Instead, we plot an "evolving average", whereby the result for query n indicates the average value for all queries up to and including n. This allows to compare the degree of increase or decrease in recall and time at each point, i.e., for each query. Interestingly, we see that the store can sometimes return better recall than the hybrid approaches, as happens for query 27, where the interim bindings returned by OpenLink cannot be dereferenced. However, the recall is improved by the hybrid approaches for the subsequent queries. It is further interesting to observe that the hybrid approaches seem to be "grouped", i.e., `coh-best`, `coh-2` and `sel-best` show very similar performance over all queries, while `coh-incoh` first "follows" `sel-thr` and others, but performs similar to `sel-2` for later queries. The evolving average of the query times in Figure 11 basically confirm the results from Figure 9.

In summary, we found that sending more patterns live with fewer bindings (as with the selectivity-based ordering) is in parts faster than sending fewer patterns live with more bindings (coherence-based ordering). Though selectivity *ordering* does not consider coherence, a coherence-based split ensures that more of the query is executed live to compensate. The lower query times suggested by the coherence-based orderings are often compensated by the low selectivities of operators in the lower levels of query plans. Still, as a guideline, if the objective is to maximise recall, one should choose coherence-based ordering, which is still faster than the live approach. If one is willing to sacrifice some recall for even faster results, then selectivity-based ordering is a good choice. However, the performance of the selectivity-based approaches seems to heavily depend on the actually chosen split strategy. Generally, the question of how to pick the

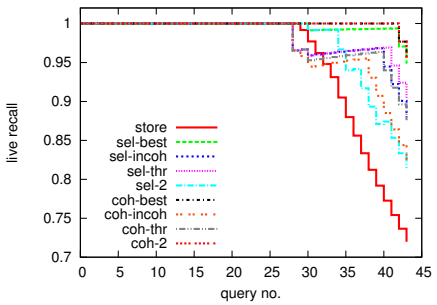


Fig. 10. Evolving average for *query recall* with different order and split strategies

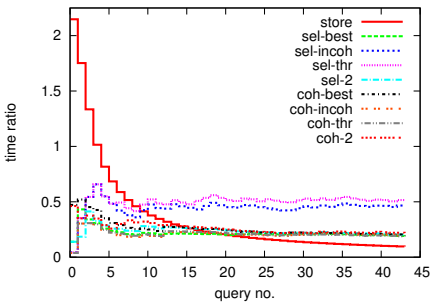


Fig. 11. Evolving average for *query time* with different order and split strategies

split position cannot be ultimately answered without taking the actual query and other characteristics into account. While the actual value of the coherence threshold did not have an impact as high as expected, we could show that the coherence estimates themselves are of great benefit, especially for ordering.

7 Conclusion

Based on an empirical study showing that popular public SPARQL stores struggle to maintain coherent cached indexes of Linked Data, we propose a hybrid query architecture that aims to combine the best from centralised indexes and novel live querying approaches. We discussed extracting coherence measures from centralised endpoints based on probing queries, and showed that they can be combined with reordering and hybrid-split strategies to design an effective hybrid query plan that speeds up live results while freshening up centralised results.

Still, some open questions remain. We have looked at a wide variety of configurations, which hint at the potential complexity of hybrid query planning. More complex cost models—including, e.g., the potential for multiple splits—may reveal novel optimisations that we have not considered herein, further pushing the boundaries of fresh vs. fast results. Furthermore, we can only estimate the accuracy of store results using coherence estimates; other mechanisms that cross-check the sources of data (i.e., the named graphs from which the store computes answers) against their current versions could yield yet more accurate statistics. Also, we consider the live and centralised query components to be strongly decoupled. However, the store may serve as a source selection index to enhance the live results given by a zero-knowledge approach such as LTBQE.

Given the potential scope and dynamicity of Linked Data, query engines will need to employ a range of techniques to efficiently offer fresh results with broad coverage. We believe that our hybrid query approach makes a significant step in this direction by combining results from centralised and decentralised query engines. Still, we may only have scratched the surface of what is possible.

References

1. C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and optimization of the SPARQL 1.1 Federation extension. In *ESWC*. Springer, 2011.
2. B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. FactForge: A fast track to the web of data. *SWJ*, 2(2):157–166, 2011.
3. C. Bizer, A. Jentzsch, and R. Cyganiak. State of the LOD Cloud (v0.3). Online report, FUB/DERI, 2011.
4. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*. Springer, 2009.
5. O. Görlitz and S. Staab. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*. Springer, 2011.
6. O. Hartig, C. Bizer, and J. C. Freytag. Executing SPARQL queries over the web of Linked Data. In *ISWC*. Springer, 2009.
7. M. Karnstedt, K. Sattler, I. Geist, and H. Höpfner. Semantic Caching in Ontology-based Mediator Systems. In *Berliner XML-Tage*. XML-Clearinghouse, 2003.
8. G. Ladwig and T. Tran. Linked Data query processing strategies. In *ISWC*. Springer, 2010.
9. G. Ladwig and T. Tran. SIHJoin: Querying remote and local Linked Data. In *ESWC*. Springer, 2011.
10. Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *ISWC*. Springer, 2010.
11. S. Podlipnig and L. Böszörményi. A survey of Web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.
12. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*. Springer, 2008.
13. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: A federation layer for distributed query processing on linked open data. In *ISWC*. Springer, 2011.
14. M. Stocker and A. Seaborne. ARQo: The architecture for an ARQ static query optimizer. Technical report, HP Labs Bristol, 2007.
15. H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *WWW*. ACM, 2004.
16. T. Tran, L. Zhang, and R. Studer. Summary models for routing keywords to Linked Data sources. In *ISWC*. Springer, 2010.
17. G. Tummarello, R. Delbru, and E. Oren. Sindice.com: Weaving the open linked data. In *ISWC/ASWC*, pages 552–565. Springer, 2007.
18. J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. Towards dataset dynamics: Change frequency of linked open data sources. In *LDOW*. CEUR, 2010.
19. J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Improving the recall of live Linked Data querying through reasoning. In *RR*. Springer, 2012.
20. J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over Linked Data. *WWWJ*, 14(5–6):495–544, 2011.
21. J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira. Freshening up while staying fast: Towards hybrid SPARQL queries. In *EKAW*. Springer, 2012.
22. J. Umbrich, M. Karnstedt, J. X. Parreira, A. Polleres, and M. Hauswirth. Linked Data and live querying for enabling support platforms for Web dataspaces. In *DESWEB Workshop, ICDE*. IEEE Computer Society, 2012.
23. G. T. Williams and J. Weaver. Enabling fine-grained HTTP caching of SPARQL query results. In *ISWC*. Springer, 2011.