Mobile Semantic Query Distribution with Graph-Based Outsourcing of Subqueries

William Van Woensel

NICHE Research Group, Faculty of Computer Science,
Dalhousie University, Halifax, Canada
{william.van.woensel}@dal.ca

Abstract. While mobile computing domains have illustrated the usefulness of mobile semantic data, improvements in mobile hardware are paving the way for local semantic data access. To support this, a number of tools have been developed for storing, querying and reasoning over local semantic data. However, recent benchmarks have shown that mobile hardware still imposes limitations on efficient local data querying. Additionally, mobile scenarios pose unique challenges due to their dynamic nature; making it difficult to replicate semantic data a priori for local querying. In this paper, we propose a graph-based query distribution approach, which efficiently distributes query execution across configured remote datasets. Importantly, our approach aims to identify subqueries that can be outsourced to remote datasets, thus reducing local joining work.

Keywords: mobile applications, query distribution, resource constraints

1 Introduction

As shown by the Linked Open Data cloud [1], a staggering number of online, machine-readable and interconnected Semantic Web datasets are currently available. Multiple tools and techniques have been developed to access this wealth of data. Local replication [2] involves replicating relevant parts of semantic datasets locally, allowing for robust and efficient access. Virtual data integration, or query distribution, distributes queries over the remote datasets themselves, integrating the results locally [3, 4].

For some time now, mobile devices have met the hardware requirements for managing and querying Semantic Web data. Reflecting this evolution, various mobile computing domains currently leverage semantic data, including augmented reality [5], recommender systems [6], location-aware [7] and context-aware systems [8], mobile tourism [9] and m-Health [10]. Supporting these approaches, multiple tools have been developed for constructing, managing, querying and reasoning over local semantic data on mobile devices, including AndroJena [11], a port of the well-known Apache Jena framework [12], and Rdf On The Go [13], which was specifically developed for mobile systems. However, as shown by recent benchmarks [14, 15], mobile hardware limitations regarding processing power, memory and battery capacity, limit the scale of purely local solutions. Many mobile scenarios also pose unique challenges due to their highly dynamic nature; e.g., cases where semantic data related to the user's dynamic

context needs to be continuously accessible. Such scenarios makes a priori, local replication of relevant data on the device problematic. Virtual data integration solutions bypass this issue by executing queries directly on remote datasets. Moreover, by leveraging the capabilities of remote datasets, opportunities exist for dealing with mobile hardware limitations.

In particular, subqueries may be outsourced to relevant remote datasets, relieving the mobile client of join processing. We also note that server hardware hosting these datasets are better equipped, both hardware-wise and regarding data access optimizations (e.g., join indices), to execute these subqueries to begin with. Moreover, less intermediate results are returned to the device, reducing bandwidth usage. To allow identifying subqueries that are resolvable by a particular dataset, we propose indexing graph patterns (i.e., graph structure with only predicate edges) found in the dataset. For a given query and set of configured datasets, suitable subqueries are found by determining subgraph isomorphism between the query subgraphs and dataset graph patterns. Although subgraph checking is an NP-hard problem, it has reasonable execution times for many real-world scenarios and is often used in graph databases [16].

In this paper, we present a graph-based, semantic query distribution approach, which outsources suitable subqueries via graph pattern indexing and matching. We apply a custom, back-tracking subgraph isomorphism algorithm, which is able to identify subqueries suitable to be executed on a particular remote dataset. We present an evaluation comparing our system to a predicate-based approach, using a real-world dataset.

Section 2 discusses the indexing of dataset graph patterns. Section 3 presents our query distribution approach. Section 4 shows an initial evaluation of our approach, while Section 5 discusses related work. Section 6 presents conclusions and future work.

2 Indexing dataset graph patterns

To identify dataset graph patterns, our system first collects instance RDF graphs. Duplicate instance graphs and subgraphs are hereby ruled out by applying subgraph checks on the collected instance graphs, leaving only distinct graph patterns. Below, we show the pseudocode for this indexing step:

```
1.  q ← SELECT * WHERE { ?s ?p ?o . }
2.  result ← execute(q, dataset)
3.  graphs ← ∅
4.  for each t1 in result
5.    result = result − t1
6.    graph1 ← ∅
7.    triples ← [ t1]
8.    for each curT in triples
9.      graph1 ← graph1 + curT
10.     for each t2 in result
11.       if matches(curT, t2) then
12.         result ← result − t2
13.         triples ← triples + t2
14.       end if
15.     end for
16.   end for
17. end for
18. exists ← false
19. for each graph2 in graphs
20.   if is_subgraph(graph1, graph2) then
21.     exists ← true
22.     break
23.   else if is_subgraph(graph2, graph1) then
24.     graphs ← graphs − graph2
25. end for
26. if ! exists
27.   graphs ← graphs + graph1
```

Code 1. Algorithm for extracting dataset graph patterns.

Lines 1-2 obtain all triples from the dataset. For each distinct result triple (lines 4-5), a new graph pattern is created (line 6), as well as a list of candidates for expansion, initially containing the result triple (line 7). Each candidate for expansion is added to the graph pattern (lines 8-9), and other triples linking to the current candidate (lines 10-11) are themselves added as candidates for expansion (line 13). This process continues until no more new expansion candidates are found, meaning a disjoint instance RDF graph has been identified.

Subsequently, the algorithm checks whether the collected graph is a subgraph of another, previously identified graph, or vice versa (lines 18-27). In case it is found to be a (non-proper) subgraph, the newly found graph is ignored (lines 20-22). In case a previously indexed graph is a subgraph of the new graph, the previous is removed & the new graph is added (lines 24-25, 26-27). Else, the new graph is added to the index as a new graph pattern (lines 26-27).

In Section 3, we elaborate on the implementation of the *is_subgraph* function. We note that the *matches* function only considers certain types of links, to maximize the re-use of extracted graph patterns. Overall, the function may consider 4 links to extend an instance graph, as illustrated in Figure 1:

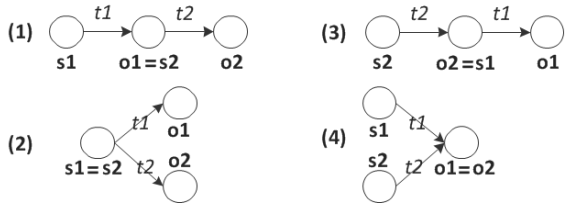

Figure 1. Potential links followed during indexing.

Two triples may be considered part of the same instance graph in case they represent a path (links (1) and (3), and if they share the same subject (link (2)) or object (link (4)). In practice however, we found that considering link (4) typically leads to cases where only a single (huge) graph pattern can be extracted (i.e., about the same size as the dataset). For instance, most resources will often be typed with *owl:Thing*, resulting in only one instance graph. Currently, we follow a pragmatic solution to this problem, by simply ruling out link (4); thus maximizing re-use of dataset graph patterns, and significantly reducing the size of extracted graph patterns. On the other hand, we note that this will lead to problems if the shared object itself is involved in other triples as subject. Tackling this issue more effectively is considered future work.

After extracting the graph patterns, they are added to an index keeping the graph patterns for each dataset. Ideally, the resource-intensive indexing process occurs on the dataset server, ruling out the need to communicate the entire dataset to another location. Subsequently, indexed graph patterns are communicated to the mobile systems, and updated each time significant changes occur that alter the previously indexed patterns.

3 Graph-based Query Distribution

Based on the dataset graph pattern index (see Section 2), query execution will be distributed across matching datasets. To cope with mobile device limitations, our main goal is to distribute coherent subqueries to relevant datasets, thus outsourcing the resource-intensive join work to

database systems better equipped to execute the work, both hardware-wise and regarding internal optimizations (e.g., join indices).

We apply a graph-based approach to identify subqueries resolvable by remote datasets. To that end, we developed a custom subgraph isomorphism algorithm, discussed in Section 3.1. Next, we present our query distribution algorithm (Section 3.2).

3.1 Backtracking subgraph-isomorphism

A subgraph isomorphism algorithm checks whether graph g_1 is isomorphic to a (non-proper) subgraph of graph g_2. To suit our purposes, we developed a slightly modified algorithm, which is able to identify the largest (non-proper) subgraph of g_1 that is isomorphic to a (non-proper) subgraph of g_2. In doing so, our system can identify which subqueries, or query subgraphs, are resolvable by dataset subgraphs.

For this purpose, a listener traces the algorithm execution, and tracks the most successful comparison. Furthermore, certain optimizations, applicable when only checking for subgraph isomorphism, need to be dropped (indicated by (†))[1]. Code 2 shows the pseudocode for this algorithm. In the *is_subgraph* function, lines 2-5 check whether the g_1 graph is subgraph-isomorphic to graph g_2. In particular, for each combination of nodes n_1 and n_2, the code checks whether the graph reachable from n_1 is a subgraph of the n_2 graph, using the *compare* function. In case the n_1 subgraph was compared successfully to g_2 (line 5), and all g_1 nodes were mapped (line 6), g_1 is a subgraph of g_2. Else, previous mappings are undone (lines 10, 14) and another node combination is tried. If no complete match was found in any comparison, g_1 is not a subgraph of g_2 (line 20). We note that, in case no n_2 nodes were found that even partially match n_1 (line 13-16), g_1 cannot be a subgraph of g_2; and false could be returned (at line 17). However, a partial subgraph match, involving a subgraph of g_1, may still occur; so this code is left out.

```
1.  function is_subgraph(g1, g2)
2.    for each node n1 in graph g1
3.      for each node n2 in graph g2
4.        map(n1, n2)
5.        if compare(n1, n2) then
6.          if mapping_complete( )
7.            listener :: done()
8.            return true
9.          else then
10.           mapping_rollback( )
11.           listener :: compare_fail()
12.         end if
13.       else then
14.         clear_mapping(n1)
15.         listener :: compare_fail()
16.       end if
17.     end for                          (†)
18.   end for
19.   listener :: done()
20.   return false
```

```
21. function compare(n1, n2)
22.   matches ← neighbor_matches(n1, n2)
23.   if ! matches then
24.     return false
25.   end if
26.   l1: for each match_i in matches
27.     e1 ← match_i . e1
28.     l2: for each e2 in match_i . e2s
29.       listener :: comparing(e1, e2)
30.       map ← get_mapping(e1 . toNode)
31.       if map = NULL
32.         map(e1 .toNode, e2 .toNode)
33.         if compare(e1.toNode, e2.toNode)
34.           break l2
35.         else clear_mapping(e1 .toNode)
36.         end if
37.       else if map = e2 .toNode
38.         break l2
39.       end if
40.     end for                          (†)
41.   end for
```

[1] These optimizations are enabled while building the dataset graph pattern index.

```
42.  listener :: done_comparing(n1, n2)
43.  return true
```

Code 2. Custom, back-tracking subgraph isomorphism algorithm.

The *compare* function starts by checking for overlaps between the outgoing edges of n_1 and n_2 (*neighbor_matches*; line 22). In case n_1 is not a leaf node and no overlaps are found, false is returned (lines 23-25). For each overlap, the function checks whether the e_1 to-node is already mapped to a g_2 node (line 30; to avoid infinite loops). If so, and if it was already mapped to the e_2 to-node, edge e_2 matches e_1 (lines 37-38). If not, the e_1 to-node is mapped to the e_2 to-node (line 32), and the function recursively compares these two to-nodes (line 33). E.g., in case no overlapping edges are found, this call will return false; if n_1 turns out to be a leaf node, it will return true. In case n_1 and n_2 recursively match, edge e_2 matches e_1 (line 33-34). If not, the previously assigned mapping is removed (line 35) and another edge e_2 (if any) is tried[2]. Again, at this point, the algorithm could return false if no matches are found for e_1 (at line 40), since all e_1 edges need to be matched for a subgraph match. However, to allow identifying partial matches (i.e., involving a subgraph of g_1), all e_1 edges need to be tried; even if some have already failed.

The *neighbor_matches* function returns true in case n_1 is a leaf node; since this means the n_1 subgraph has been checked completely. Else, it collects the overlaps between the outgoing edges of e_1 and e_2: whereby two edges match in case both of them have the same label; either of them represents a variable; or the e_1 label represents a subproperty of e_2. If no matches are found for any edge e_1, the function returns false; else, it returns the overlapping edges.

To track the largest matching subgraph of g_1, a listener is notified when two edges are being compared (line 29), when two nodes are finished comparing (line 42), when a g_1 node comparison failed (lines 11, 15) and when comparison is done (lines 7, 19). Upon finishing the subgraph comparison, the listener records the match by assigning dataset associated with the dataset edge to its matching query edge; together with an ID uniquely identifying the subgraph comparison[3].

3.2 Query Distribution

To achieve virtual data integration, query execution is distributed across the configured datasets, and the results integrated locally. In its simplest form , this involves splitting up a query into its smallest units (i.e., triple patterns), executing them on each individual dataset, and combining the results. In doing so, a query distribution system ensures that all results are returned, even for queries that are not resolvable by any single dataset. Initially, such a *Query Distribution Plan (QDP)* consists of n^m *query sets*, each representing a particular result integration:

$$QDP = \left[\left\{ t_1 \rightarrow D_x, \dots, t_i \rightarrow D_y, \dots, t_m \rightarrow D_z \right\}, \dots\right]$$
$$n = size(datasets), m = size(query), 0 < i < m, 0 < x, y, z < n$$

Formula 1. Query Distribution Plan (QDP)

[2] Multiple e_2 matches for e_1 are possible, and vice-versa (e_2 is only matched to one e_1 at a time).
[3] This unique ID is required by the query distribution algorithm (see Section 3.2).

Where $t_i \rightarrow D_x$ stands for an atomic subquery, i.e., executing a single triple pattern t_i on dataset D_x; a set of subqueries between accolades forms a query set, standing for a particular integration of results; and the set of query sets make up the QDP, standing for all possible result integrations.

In our query distribution approach, graph patterns from an incoming query are compared to the set of dataset graph patterns (see Section 2), using subgraph isomorphism checks. After these checks, matching datasets are assigned to the query graph edges (see Section 3.1, last paragraph), indicating which query triples (each corresponding to an edge) are collectively resolvable by particular datasets. Based on these results, given a query set, multiple t_i matched to the same D_y (during the same subgraph check) can be grouped into the same subquery:

$$query\ set = \{t_1 \rightarrow D_x, \dots t_{i,j,k} \rightarrow D_y, \dots, t_m \rightarrow D_z\}$$
$$n = size(datasets), m = size(query), 0 < i, j, k < m, 0 < x, y, z < n$$

Formula 2. Grouping subqueries in query sets based on their shared dataset.

In the query shown in Figure 3, subqueries $t_1, t_2, t_3 \rightarrow A$, $t_2, t_4 \rightarrow B$, $t_1 \rightarrow C$ and $t_5 \rightarrow D$ can be distinguished into their respective query sets.

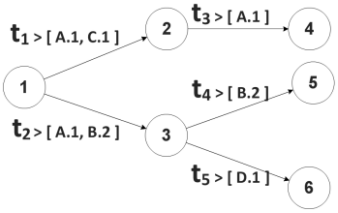

Figure 3. Example query matched to the configured datasets.

In this process, it is important to consider the particular subgraph check in which the matching dataset was found. For instance, consider the following cases:

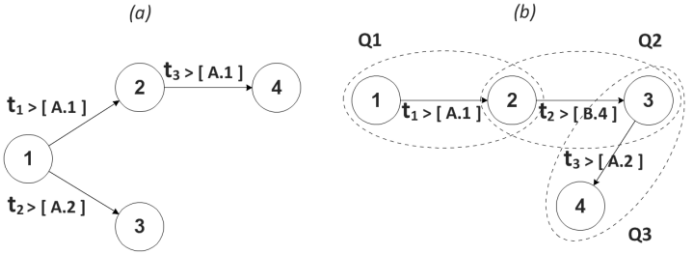

Figure 4. Distinct graph matches when constructing subqueries.

In case (a), part of the query (t_1, t_3) was matched to a particular graph pattern (1) from A during one subgraph check, while the remainder (t_2) was matched to a different graph (2) from A during another check. However, these two dataset graph patterns are disjoint; no single instance graph exists that covers both graph patterns[4]. As such, this particular query set will never yield any results, and need to be removed from the QDP. Case (b) illustrates that this reasoning is only valid when considering *connected* query triples (i.e., with shared variables).

[4] Else, they would have been combined during the graph extraction process (see Section 2).

Here, an intermediate query triple t_2 is executed on dataset B, which may yield results that connect t_1 with t_3; in other words, an instance graph, integrated from both datasets, may exist that connects all 3 query triples.

Further, we note that triple patterns executed on the same dataset, but not sharing any variables, should ideally be kept separate. Putting these into the same subquery will lead to a Cartesian product, resulting in a huge number of results returned by the remote dataset; whereas the associated local computational work is comparatively low.

Below, we show the pseudocode for post-processing the QDP, based on subgraph matches:

```
1.  l1: for each query set in qdp
2.    subqueries ← group on dataset and shared vars (query set)
3.    for subquery in subqueries
4.      l2: for each t1 in subquery
5.        for each other t2 in subquery
6.          if t1 .dataset .id ≠ t2 .dataset .id then
7.            remove query set from qdp
8.            break l2
9.          end if
10.     end for
11.   end for
```

Code 5. Processing the QDP based on subgraph matching results.

For each query set, query triples are grouped into subqueries based on assigned dataset and shared variables (lines 1-2). If one of these subqueries involves two query triples, assigned to the same dataset but associated with a different dataset graph pattern (lines 4-6), the query set is removed from the QDP (lines 7-8).

After generating a QDP, it is passed to the execution engine. For each subquery, the engine creates and executes a SPARQL query on the associated remote dataset. To integrate subquery results from a single query set, we apply a hash join. Results from multiple query sets are combined via a union operation. Since the same subquery-on-dataset combination will occur in multiple query sets (see Formula 1), the engine caches previous results for later re-use.

4 Evaluation

This section presents a preliminary evaluation of our query distribution approach. In our evaluation, a client app poses a query that requires data from two datasets to be integrated. To illustrate the usefulness of graph-based query distribution, we compare our approach to a straightforward predicate-based approach, which distributes incoming queries solely based on query predicates and indexed dataset predicates. For each query triple, the approach checks which datasets contain its concrete predicate; and then executes the query triple (potentially grouped in a subquery) on the found datasets.

Below, we elaborate on the evaluation setup, including current implementation components. Then, we discuss the results of each query distribution approach.

4.1 Setup

We ran all experiments 10 times, and took the average of the performance times. Below, we elaborate on other relevant aspects:

- **Dataset & query**

- *Dataset*

Using interlinks made available by DBPedia[5], we extracted two small datasets from DBPedia (3846 triples; 487 Kb) and Geonames (1210 triples; 157 Kb). Both datasets supply different data on the same resources; whereby the extracted Geonames dataset uses DBPedia resource URIs to allow for data integration. Both datasets can be found online [19]. Although these datasets are relatively small, we will show that these a) already result in non-trivial execution times and b) indicate significant differences in performance between the evaluated approaches. Respectively, 7 and 2 distinct graph patterns were found in the extracted DBPedia and Geonames datasets.

- *Query*

Our evaluation executes the following query, selecting the label, type, coordinates and website of geographic entities (namespaces omitted for brevity):

```
1. SELECT * WHERE {
2.   ?subject rdfs:label ?label .
3.   ?subject rdf:type ?type .
4.   ?subject wgs:lat ?lat .
5.   ?subject wgs:long ?long .
6.   ?subject dbp:website ?website . }
```

Code 6. Evaluation query.

This query returns 51 results on the integrated dataset.

- **Hardware**

- *Dataset*

Both datasets were made accessible using the Apache Fuseki [17] SPARQL server, deployed on a Dell PowerEdge 2950 Server running Windows Server 32-bit, with (2) Intel Xeon 2.33 GHz and 64 Gb RAM. The datasets were indexed on a Lenovo Thinkpad, running Windows 7 64-bit, with Intel Core i7-3520M 2.90 Ghz and 8Gb RAM.

- *Mobile*

We used a LG Nexus 5 (model LG-D820) running Android 5.1.1 (Lollipop), with 2.26 GHz Quad-Core Processor, 2Gb RAM and 32Gb storage. The mobile device connects to the SPARQL server over an Internet connection[6] (using WiFi).

- **Libraries:** to index datasets, we utilize Apache Jena 2.11.0 [12]. For performing query distribution on mobile systems, we rely on AndroJena 0.5 [18].

[5] Such interlinks indicate resource equivalence with other major datasets.
[6] To mimic real-life conditions, the SPARQL endpoint was not hosted on the same local network.

4.2 Results

Figure 5 shows the evaluation query graph after matching to indexed graph patterns, together with the resulting query sets. Although the DBPedia dataset (DB) contains individual triples matching all query triples, no single connected instance graph connects all 5 query triples (see DB.1 and DB.2 graph matches). Therefore, resolving this query requires data integration with the GeoNames dataset (GN). Based on graph matches shown in the query graph, two query sets are generated, each with a coherent subquery assigned to one of the datasets.

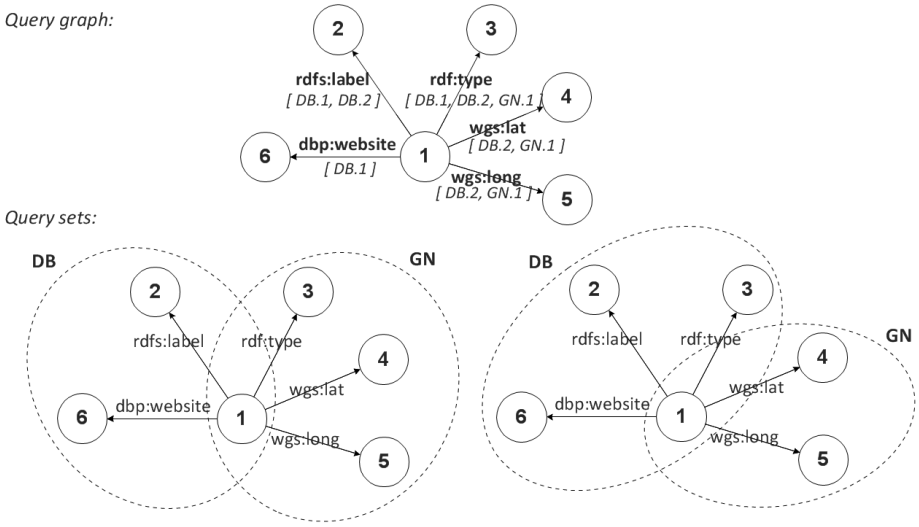

Figure 5. Query graph & subqueries for evaluation query.

For the predicate-based approach, we consider two configurations: a configuration where joins are not outsourced (*no-outsource*); and a configuration where, per query set, query triples assigned to the same dataset are grouped into the same subquery (*outsource*). In Table 1, we indicate the number of query sets, remote query executions and total number of individual query results to be joined.

	graph-based	**predicate-based**	
		no-outsource	*outsource*
# query sets	2	8	8
# query exec.	4	8	15
# indiv. results	151	1030	451

Table 1. Number of query sets and remote query executions.

The overall number of query executions is relatively low, since an internal cache is kept to avoid re-sending the same subquery (see discussion after Code 5). We also note that, when outsourcing queries, additional subqueries will be constructed. Therefore, the potential for reusing cached results is reduced, and the overall number of query executions is comparatively

increased (see *predicate-based* > *outsource* column). However, the local join work is reduced, as illustrated by the total number of individual query results.

Table 2 shows the performance results, where ID stands for identifying relevant datasets and QDP for constructing the dataset (see Code 5):

	graph-based	**predicate-based**	
		no-outsource	*outsource*
parse query	8		
ID	4	0.1	0
QDP	40	0	0.7
execute	529	2065	1856
join	16	1654	19
total	597	3728	1883

Table 2. Performance results (ms).

Since less queries are sent to the datasets, executing queries takes much less time for *graph-based*. Since *pred-based* > *outsource* and *graph-based* both outsource join work to the remote dataset, locally joining results is much faster as well. Despite its extra overhead when identifying relevant datasets and constructing the QDP, our graph-based query distribution approach outperforms either predicate-based approach.

Creating the graph index and predicate index takes ca. 4431ms and 80ms, respectively. We note that that the graph creation process only needs to be applied in case the dataset contents are updated significantly, causing a change in its graph patterns.

5 Related work

The Distributed ARQ (DARQ) [3] and Semantic Web Integrator and Query Engine (SemWIQ) [4] systems keep an index with summary dataset info. DARQ keeps so-called service descriptions, including found predicates, constraints on subjects and objects occurring with these predicates, and statistical data. The SemWIQ system maintains a catalog per data source, which keeps a list of classes and their number of instances, as well as a list of properties and their number of occurrences. Given a posed query, these indices are used to determine which triple patterns should be sent to which datasets. As such, these works do consider join outsourcing; which has the potential for large performance gains, as shown by our evaluation.

The approach in [20] resembles our work, as it focuses on indexing found predicate sequences or paths. This allows identifying datasets that can handle particular query predicate paths, with the goal of reducing local join work. In contrast, our approach supports outsourcing any kind of subquery, and is not just limited to path-based queries.

6 Conclusions & Future work

In this paper, we presented a graph-based query distribution approach, focusing on outsourcing subqueries to relevant remote datasets. We presented a mechanism for indexing graph patterns in remote datasets; a custom, backtracking subgraph isomorphism algorithm; and our graph-based query distribution mechanism. Our evaluation shows that our approach has the potential to significantly reduce the number of queries to be sent to remote datasets, as well as minimize the resulting local join work.

Many avenues for future work exist. By keeping summary data on graph pattern nodes (cfr. [3]), the "joinability" between graph patterns of different datasets can also be considered when ruling out query sets (see Figure 4 (b)). Edges in extracted graph patterns can be annotated with the number of associated instance graphs, to guide join optimizations. Currently, extracted graph patterns are kept per dataset. By keeping a single index, equivalent graph patterns from multiple datasets can be merged, thus reducing the number of isomorphism checks.

To allow identifying partial subgraph matches, our subgraph checking algorithm drops a number of optimizations that may result in serious performance gains. Studying other methods of efficiently determining partial query matches is future work. Furthermore, although subgraph isomorphism checking is known to be an NP-hard problem, many algorithms have been proposed over the years that solve it in a reasonable time [16]. In case our straightforward, custom algorithm leads to problematic performance for larger datasets, future work may involve studying and re-using other algorithms.

7 References

1. Cyganiak, R., Jentzsch, A.: The Linking Open Data cloud diagram, http://lod-cloud.net/.
2. Zander, S., Schandl, B.: A framework for context-driven RDF data replication on mobile devices. Proceedings of the 6th International Conference on Semantic Systems. pp. 22:1–22:5. ACM, New York, NY, USA (2010).
3. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. ESWC'08: Proceedings of the 5th European semantic web conference on The semantic web. pp. 524–538. Springer-Verlag, Berlin, Heidelberg (2008).
4. Langegger, A., Wöß, W., Blöchl, M.: A semantic web middleware for virtual data integration on the web. Proceedings of the 5th European semantic web conference on The semantic web: research and applications. pp. 493–507. Springer-Verlag, Berlin, Heidelberg (2008).
5. Reynolds, V., Hausenblas, M., Polleres, A., Hauswirth, M., Hegde, V.: Exploiting linked open data for mobile augmented reality. W3C Workshop: Augmented Reality on the Web (2010).
6. Ziegler, C.: Semantic web recommender systems. In Proceedings of the Joint ICDE/EDBT Ph.D. Workshop 2004 (Heraklion. pp. 78–89. Springer-Verlag (2004).
7. Becker, C., Bizer, C.: DBpedia Mobile: A Location-Enabled Linked Data Browser. In: Bizer, C., Heath, T., Idehen, K., and Berners-Lee, T. (eds.) LDOW. CEUR-WS.org (2008).
8. Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: Mobile Querying of Online Semantic Web Data for Context-Aware Applications. IEEE Internet Comput. Spec. Issue (Semantics Locat. Serv. 15, 32–39 (2011).
9. Keller, C., Pöhland, R., Brunk, S., Schlegel, T.: An Adaptive Semantic Mobile Application for Individual Touristic Exploration. HCI (3). pp. 434–443 (2014).

10. Puertas, E., Prieto, M.L., De Buenaga, M.: Mobile Application for Accessing Biomedical Information Using Linked Open Data. Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine. , Prague, Czech Republic (2013).
11. AndroJena, https://code.google.com/p/androjena/.
12. Apache Jena, https://jena.apache.org/.
13. Le-Phuoc, D., Parreira, J.X., Reynolds, V., Hauswirth, M.: RDF On the Go: An RDF Storage and Query Processor for Mobile Devices. 9th International Semantic Web Conference (ISWC2010) (2010).
14. Patton, E.W., McGuinness, D.L.: A Power Consumption Benchmark for Reasoners on Mobile Devices. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C.A., Vrandecic, D., Groth, P.T., Noy, N.F., Janowicz, K., and Goble, C.A. (eds.) The Semantic Web - {ISWC} 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part {I}. pp. 409–424. Springer (2014).
15. Woensel, W. Van, Haider, N. Al, Ahmad, A., Abidi, S.S.R.: A Cross-Platform Benchmark Framework for Mobile Semantic Web Reasoning Engines. The Semantic Web - {ISWC} 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part {I}. pp. 389–408 (2014).
16. Lee, J., Han, W.-S., Kasperovics, R., Lee, J.-H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proceedings of the 39th international conference on Very Large Data Bases. pp. 133–144. VLDB Endowment (2013).
17. Apache Fuseki, http://jena.apache.org/documentation/fuseki2/.
18. AndroJena, http://code.google.com/p/androjena/.
19. Online Datasets, https://niche.cs.dal.ca/materials/qd/.
20. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.: Towards distributed processing of RDF path queries. Int. J. Web Eng. Technol. 2, 207–230 (2005).