Transparent Mobile Querying of online RDF sources using Semantic Indexing and Caching

William Van Woensel[1], Sven Casteleyn[2], Elien Paret[1], Olga De Troyer[1]

[1] Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
{William.Van.Woensel, Elien.Paret, Olga.Detroyer}@vub.ac.be
[1] Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
Sven.Casteleyn@upv.es

Abstract. Due to advancements in mobile technology and connectivity, mobile devices have become fully-fledged web clients. At the same time, more and more Semantic Web data is becoming available, to a point where it becomes usable for various mobile application scenarios. However, most applications are limited to using pre-defined query endpoints in order to access Semantic Web data, which leaves a huge part of the Semantic Web, consisting of online RDF files and semantically annotated websites, inaccessible. In this paper, we present a mobile query service for the efficient and transparent querying of large amounts of small online RDF sources. In order to achieve this, the query service relies on two key components: 1/ a lightweight semantics-based indexing scheme, to identify sources relevant to posed queries, and 2/ a caching mechanism that locally stores frequently-used data.

1. Introduction

The Semantic Web has evolved greatly in its ten-year existence. With the building blocks (i.e. RDF(S), OWL and SPARQL) already in place for some years, and mature and reliable software becoming widespread (e.g. Jena, Sesame, Virtuoso), we are now finally seeing increasing amounts of semantic web data becoming available online. The Linked Data initiative has fostered the deployment and interconnection of large semantic datasets, covers various domains and currently amounts up to 4.5 trillion triples. The so-called lightweight Semantic Web, where existing (X)HTML content is annotated using semantic annotation languages (e.g. RDFa, microformats), is also growing. According to Yahoo! BOSS, currently close to 955 million websites make use of RDFa. These RDFa annotated websites represent semantic sources in their own right, as RDF triples can be extracted from their annotations.

In a parallel evolution, the performance and screen resolution of mobile devices has steadily increased, up to a point where they are capable of supporting common software applications (e.g. organizer, mail client, Web browser). Combined with the widespread availability of wireless networks and affordable high-speed transmission rates for mobile phone networks, these mobile devices have become fully-fledged, (quasi-)permanently connected Web clients. Their use has become prevalent, and it is estimated that mobile Internet access is to surpass desktop access by 2014[1].

Therefore, a key factor in the realization of the Semantic Web is realizing efficient *mobile* access to its data. In various mobile application settings, the management, access and integration of local and remote semantic data is already of paramount importance. Examples include context-aware service discovery [1], mobile augmented reality [2], mobile personalization [3], context-aware systems [4] and mobile social applications [5]. Most existing mobile Semantic Web applications gain access to remote RDF(S) datasets via their exposed SPARQL query endpoints, allowing efficient access to online datasets without straining the mobile device. However, it requires a considerable effort to set up a query endpoint, as none of the existing solutions (e.g. OpenLink Virtuoso, Sesame Server) work out-of-the-box and require substantial setup time. In practice, only major data providers make query endpoints available; smaller online RDF sources are mostly put online as semantic documents (i.e. RDF files). Sindice, a lookup index for the Semantic Web, currently indexes around 246 million of such online semantic documents. This means that a huge part of the Semantic Web, consisting of online RDF files and semantically (RDFa) annotated websites, is currently unavailable to mobile clients.

We present a client-side query service that can be employed by mobile applications to transparently and efficiently query large amounts of small online RDF sources. This service is built on top of an existing mobile query engine (such as androjena[2] or RDF On the Go [6]) to locally query RDF(S)/OWL data. In order to achieve efficient access on mobile devices, with limited processing power and storage space, the query service relies on two key components: 1/ a lightweight indexing scheme, to identify sources relevant for a particular query based on semantic source metadata (i.e. occurring predicates, subject and object types), and 2/ a caching mechanism that locally stores frequently-used data. We evaluate different variants of each component, and discuss their respective advantages and disadvantages. We also compare the performance of the query service to that of the native mobile query engine.

2. General approach

As mentioned in the introduction, we focus on providing query access to the huge set of online RDF files and semantically annotated websites. Evidently, it is not possible to consider the entire dataset in existence. Instead, we focus on a selection of this data, as typically required by a certain type of mobile applications, such as semantic context-aware systems [4] or mobile social applications [5]. This querying scenario has its own set of challenges for efficient access. One challenge is to identify sources from this dataset that contain information relevant for a posed query, with a high degree of selectivity. This way, sources irrelevant for the current query can be excluded, keeping the final dataset to query smaller and manageable, and therefore

[1] http://www.morganstanley.com/institutional/techresearch/pdfs/Internet_Trends_041210.pdf
[2] http://code.google.com/p/androjena/

the overall query execution time lower. To achieve this, we build and maintain an index containing metadata about datasources. A second challenge is that, once obtained, source data should be cached locally for efficient access and later re-use. The caching of data is paramount in our setting, because of the overhead of obtaining query-relevant data; every datasource that may contain some query-relevant information needs to be downloaded in its entirety. Naturally, the amount of cached data will be limited due to the space restrictions on mobile devices.

Our approach consists of two phases: the indexing phase and the query phase. An overview of these two phases can be found in fig. 1. In the indexing phase, references to new online RDF sources are received from the application (a.1). In our evaluation (see section 5), this application is a context-aware component called SCOUT [4], which identifies datasources related to physical entities in the user's vicinity (e.g. by reading URLs from RFID tags near the entities). The Source Manager downloads these sources, and extracts metadata on their content (a.2). Our approach focuses on semantic metadata, namely used predicates, subject and object types. This metadata is then added to our index, called the Source Index Model (SIM), along with the URL of its origin source (a.3). Finally, once the source has been indexed, it is passed to the cache component, which locally caches the source data (a.4).

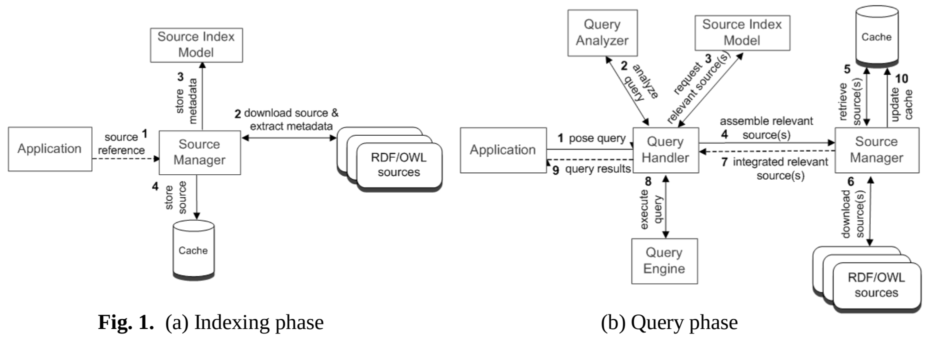

Fig. 1. (a) Indexing phase (b) Query phase

The query phase is triggered whenever the application poses a new query to the query service (b.1). Firstly, the Query Handler analyzes the query and extracts metadata, i.e. used predicates, subject and object types, from the query (b.2). This query metadata is passed to the SIM, which matches it to the extracted metadata of the indexed sources, and returns references to the sources containing query-relevant information (b.3). Based on these source references, the Source Manager then obtains the required source data (b.4), ideally retrieved from local cache (b.5), or else by re-downloading the corresponding online sources (b.6). Once all source data is obtained, it is combined in one dataset (b.7) on which the query is executed (b.8). After query execution, the results are returned (b.9) to the application. Finally, the cache is updated with the downloaded source data (b.10). In the following sections, we elaborate on the two major parts of our approach, namely indexing and selection of relevant of sources, and the local caching of source data.

3. Indexing and selecting sources

Our practical experience (comfirmed in our experiments, see section 5) shows that executing a query over large sets of datasources is problematic on a mobile device. For one, querying very large datasets slows down query execution time. Also, the entire dataset to query needs to be kept in memory for fast querying, leading to out-of-memory exceptions for even relatively small amounts of sources (around 400 sources with total size of 67Mb). As a result, we aim to keep the dataset to be queried (called the query dataset) as small as possible. We achieve this by keeping metadata from online datasources in the Source Index Model (SIM); posed queries are analyzed and matched to this source metadata, allowing us to rule out sources irrelevant to the query. Afterwards, the query is executed on the reduced dataset.

Our indexing mechanism needs to comply with certain requirements, related to our mobile, volatile setting. Because we are dealing with mobile devices, we must firstly minimize the required storage space and computational effort for the index. Additionally, in our mobile setting, the query service may receive references to new datasources at any time (e.g. a context-aware system passing data related to the surroundings of the user), meaning the index must be updateable on-the-fly. Therefore, we require a lightweight index that is quick to construct and maintain, and requires minimal space (for optimal performance, the index should fit into volatile memory). At the same time, the index should still guarantee a high selectivity (in other words, filter large amounts of irrelevant sources), in order to reduce the query dataset. We elected for a compact index storing semantic metadata on datasources; namely, the found predicates together with their subject/object types[3]. This metadata can be efficiently obtained and does not require a lot of storage space (compared to other types of metadata, see related work), making it lightweight. Furthermore, predicates and subject/object type restrictions are often specified in queries, allowing for a high source selectivity (as also shown by our evaluation, see section 5). Therefore, it also complies with the second requirement.

During our evaluation, we compared three variants of the SIM: the first variant (SIM1) stores the found predicates; SIM2 stores the predicates together with their subject types; and SIM3 keeps the found predicates together with their subject and object types. Note that although the two latter variants store additional metadata, they still support queries with unspecified subject and/or object types. In our evaluation section, we compare the different SIMs to study the trade-off between source selectivity on the one hand, and computational and storage overhead on the other.

We can now make some steps in the two general phases of fig. 1 more concrete. In the first phase, the indexing phase, the required metadata is extracted using a predefined SPARQL extraction query (a.2). The metadata is then stored in one of the SIM variants, which are implemented using a collection of hashtables. In the query phase (see fig. 1b), the query is analyzed and the same kind of metadata is extracted from the query (b.2) using the SPARQL Parser library[4]. Finally, RDF triples are extracted from RDFa-annotated websites by using a ported version of java-rdfa[5].

[3] We currently consider resource types on a per-source level.

[4] http://sparql.sourceforge.net/

[5] https://github.com/shellac/java-rdfa

4. Caching of source data

The local caching of source data occurs during the indexing phase, when the source data is passed to the cache after SIM metadata extraction (see fig. 1.a.4), and at the end of the query phase, where the cache may be updated with source data in case of a cache miss (see fig. 1.b.10). Depending on the popularity of the cached data (decided by the used replacement function), some of the cache data will be kept in-memory, while other data will be stored persistently or removed. Since the metadata of removed sources is kept in the SIM, these can still contribute to a posed query; however, they will need to be re-downloaded in case they are required.

In a mobile setting, huge amounts of data can be aggregated after a while. Although it can be argued that modern mid-to high-range mobile devices have ample persistent native storage (e.g. the iPhone is available with 16 and 32 GB of storage), users are probably not keen on spending a large portion to store caching data. By applying a replacement (or removal) function, data is removed from the cache that is not likely to be required in the future. A lot of work has been done concerning replacement strategies specifically meant for location-aware systems (see related work). As our query service does not target one particular type of application, we currently rely on the generic Least Recently Used (LRU) replacement policy; further experimentation with other replacement policies is considered future work.

As was the case for the SIM, the cache needs to comply with certain requirements. Firstly, in order to reduce the query dataset and thereby the overall execution time, a sufficiently fine-grained level of storage and retrieval is needed. Secondly, only a small amount of additional data (e.g. indices such as B+-trees, hashtables) should be kept; ideally, this data should fit in volatile memory (16MB on the Android platform). This is needed to avoid frequent swapping with persistent storage that causes performance loss. Thirdly, it must be possible to add data and update the cache in a quick and efficient way, as data needs to be added on-the-fly on mobile devices with limited computational capabilities. Finally, as for any cache, we must consider the validity of the data in our cache, and ensure data freshness. In the sections below, we discuss the cache organization and cache validity in more detail, and highlight how they comply with the aforementioned requirements.

4.1. Cache organization

The source data kept in the cache can be organized in different ways, influencing the efficiency and fine-grainedness of cached data retrieval, and the cache construction and update cost. A natural choice for cache organization is to organize the cached triples via their origin source (this organization is called *source-cache)*. In this organization, a cache element (or cache unit) corresponds to a downloaded datasource. The cache contains a search index on the source URI of the cached sources, so specific sources can be quickly retrieved. This organization fulfills our second requirement (i.e. minimal additional space overhead) as only one index is kept with a relatively small amount of index entries. Also, the third requirement (i.e. efficient add and update) is met, as the source is simply stored as a single cache unit.

However, this organization does not comply with the first requirement, as it does not allow for fine-grained selection of cached source data; it returns all triples from a cached source, instead of only the triples relevant for a given query. Although such coarse-grained retrieval is unavoidable when dealing with online RDF files or RDFa annotated webpages (i.e. they can only be downloaded as a whole), this can be improved upon when dealing with locally stored data.

An alternative choice is to organize the triples according to their shared metadata (called *meta-cache*). This metadata, as was the case for the Source Index Model, includes predicate and subject/object types. For fast retrieval, indices are created on predicates, subject and object types. In this organization, a cache unit contains triples sharing the same metadata (e.g. certain predicate and subject type). This organization allows data to be obtained in a more fine-grained way, as only units with metadata matching the extracted query metadata are retrieved. As such, it complies with our first requirement, i.e. fine-grained storage and retrieval of source data. However, this comes with an additional computational overhead. Firstly, the metadata of the source triples needs to be extracted for each source; secondly, adding the extracted source data to the cache becomes more expensive, as several cache units may need to be updated for a single source (i.e. all units with metadata matching the found predicates, subject and object types). Furthermore, this organization requires three indices with considerable more index entries compared to the source-cache. Finally, each cache unit should also keep the origin source URIs of the triples it contains, to support validity checking (see following section) and to identify sources that need to be re-downloaded, in case this unit is removed and referenced again in the future (i.e. cache miss). Because of these reasons, the fulfillment of the second requirement (i.e. minimal additional storage overhead) and third requirement (i.e. efficient add and update) is tentative. However, meta-cache still has a much lower overhead than the storage and computionally demanding indices traditionally employed to speed up access to RDF data on mobile devices (see related work). We tested both cache organizations in our evaluation, checking how this computational and storage overhead compares to the reduction in data to be retrieved, combined and queried.

We can now again make some of the steps in the general two phases more concrete. For meta-cache, predefined SPARQL extraction queries are employed whenever new source data is added to the cache, in order to extract triples together with their metadata from downloaded sources. When retrieving data from meta-cache (see fig. 1.b.5,), the SPARQL Parser library is again employed to extract metadata from the query, to be matched to the metadata indices. In both source-cache and meta-cache, the indices are implementated as hashtables.

4.2. Cache validity

Extensive invalidation strategies already exist to efficiently detect invalid data items in traditional client-server architectures and mobile settings. In our setting however, the cached data originates from online RDF files (and annotated websites) stored on general-purpose HTTP webservers, instead of dedicated servers. Therefore, we need to rely the cache support features of HTTP (typically used by proxy caches) to check the validity of cache items. We utilize both the server-specified expiration times (via the `expires` or `Cache-Control: max-age` header fields) and conditional requests for files (using the `Last-Modified` header field) to verify validity of cache units.

To manage the validity of cached data, we need to keep information on the origin source of the cached information, as well as expiration information (i.e. expiration time, or last download time). In source-cache, where all data from a certain source is stored in a single cache unit, we keep this provenance and expiration information per cache unit. For meta-cache, where cache units contain triples from various sources, we also chose to store origin information per cache unit. To avoid duplicating expiration data, we keep a separate data-structure that stores the expiration information for each source. In case source data requires updating, the source is downloaded and re-indexed, replacing all outdated data from that source in the cache.

5. Experiments

Our experiments have been designed to evaluate the two major parts of our approach. For the first part, the three different SIM variants are compared. In this part, we store all downloaded datasources locally in a cache of unlimited size, to avoid cache interference. For the second part, we employ the best performing SIM variant when comparing the two cache organizations discussed in section 4.1, namely source-cache (i.e. based on origin source) and meta-cache (i.e. based on metadata).

We validate our solution in the application field of context-aware mobile applications using the SCOUT [4] framework. SCOUT gradually discovers physical entities in the user's vicinity, and offers application developers a query-able view of the user's physical environment by combining information from a variety of online RDF(S) sources describing these detected entities. To manage this data and provide efficient query access, SCOUT makes use of the developed query service. In our architecture, SCOUT fulfills the role of the application component (see fig. 1.a), gradually providing new source references to the Source manager over time.

Both SCOUT and the developed query service are based on Android OS 2.2. We employ a fine-tuned version of the androjena library to access, manipulate and query RDF data on the mobile device. The experiments were performed on a Samsung Galaxy S with a 1 GHz processor and 512 MB RAM memory. A total number of 2500 datasources is employed in the experiments[6], with total size of 477Mb and average size of around 195 Kb. These sources were partly obtained from real-life datasets (e.g. IMDB data), and complemented with generated data, using random resource types and predicates from both selected well-known ontologies (e.g. geo, travel or SUMO) and from proprietary ontologies. In order to reflect a real-world situation, different properties and types reflecting the same concepts (e.g. absolute coordinates) were randomly used. The datasources were distributed across three different web servers with different response times. Finally, we extracted five types of queries from existing mobile SCOUT applications [3, 4] to evaluate the performance of the query service. We give two examples of these queries in listing 1.

[6] The used dataset and queries can be found at http://wise.vub.ac.be/SCOUT/WISE2011/.

```
SELECT ?photo ?lat ?long
WHERE {
?statue rdf:type region:Statue .
?statue perv-sp:latitude ?lat .
?statue perv-sp:longitude ?long .
?statue dc:description ?photo .
?photo rdf:type dc-type:Image . }
```

```
SELECT ?rest ?cuisine
WHERE {
?rest rdf:type region:Restaurant .
?rest rest:typeOfCuisine ?cuisine .
?cuisine rdf:type rest:ItalianCuisine .
}
```

Listing 1. Two employed validation queries.

5.1. Indexing and selecting sources

Firstly, we measure the overhead in size and time for maintaining the SIM during the indexing phase, as the application passes source references to the Source Manager (see fig. 1.a.1). Table 1a shows the total size of the different SIM variants, for increasing portions of the total dataset (625-1250-2500 sources). The table also shows how this size compares to the total size of the indexed sources. The computational overhead to create the SIM is shown in table 1b, and includes the average times of downloading a source, extracting the required metadata, and updating the SIM.

Table 1. (a) Size overhead (Kb)

# sources&size	SIM1	SIM2	SIM3
625 (119Mb)	210 (0,2%)	412 (0,3%)	521 (0,4%)
1250 (243Mb)	416 (0,2%)	798 (0,3%)	991 (0,4%)
2500 (477Mb)	833 (0,2%)	1586 (0,3%)	1946 (0,4%)

(b) Computational overhead (ms)

	SIM1	SIM2	SIM3
extract + add	38	140	298
download	372	434	336
total	410	574	634

Table 2a illustrates the selectivity of each SIM variant for our 5 sample queries, by showing the number of identified relevant sources. For brevity, we only show the results for the total dataset of 2500 sources. In table 2b, we show the corresponding overall query execution time over the collected set of datasources. This includes the times required for query analysis, SIM access, data collection, and query execution. Note that SIM1 fails with out-of-memory error for all but the first and third queries; in the case where no SIM is used (i.e. native query engine performance) the same error occurs for any query. Therefore, no entries are available for these cases.

Table 2. (a) Source selectivity (#sources)

	SIM1	SIM2	SIM3
Q1	263	46	6
Q2	2025	326	326
Q3	48	48	4
Q4	1875	83	83
Q5	2203	328	328

(b) Query execution time (s)

	SIM1	SIM2	SIM3	no SIM
Q1	112,0	14,3	1,9	/
Q2	/	21,4	20,9	/
Q3	8,4	8,3	0,7	/
Q4	/	13,1	12,9	/
Q5	/	34,1	33,6	/

5.2. Caching of source data

Like for the SIM, we first measure the overhead of constructing and maintaining the cache during the indexing phase. For each of the tests, we allow the cache to use 75% of the size of the total dataset as persistent storage space on the device. We also allow 1Mb of volatile memory space for storing frequently-used source data (this does not include extra data such as indices). This limited amount was chosen because Android applications only have 16Mb heap space available, and other components of the query service also consume this memory: for instance, SIM3 takes up about 2Mb for 2500 sources (which will grow as the number of sources increases), while androjena graph objects are also created in-memory.

Table 3 shows the size (in Kb) of in-memory and persistent storage space used by the two different cache organizations, for increasing portions of the dataset (625-1250-2500). We show both the size of in-memory source data and the total size of the in-memory cache, which includes additional data such as index data.

Table 3. (a) size overhead for source-cache (Kb) (b) size overhead for meta-cache (Kb)

# sources	(a) in-memory		(a) persistent	(b) in-memory		(b) persistent
	total	source data		total	source data	
625	1076	1024	3083	258	136	3235
1250	1130	1024	175955	1623	871	171894
2500	1195	1024	365694	2781	1023	331431

In table 4, we show the average computational overhead of adding new source data to the cache during the indexing phase. For meta-cache, this includes extracting the metadata from the source, and updates to existing cache units. These two overheads do not occur in source-cache. For both cache types, overhead also comprises running the replacement function in case the cache becomes full.

Table 4. (a) time overhead for source-cache (ms) (b) time overhead for meta-cache (ms)

extract	add	update	replacement	extract	add	update	replacement
0	1	0	563	673	2	180	50

Tables 5 and 6 contain the overall query execution times for the query phase. The following parts are distinguished: 1) query analysis, 2) cache access, 3) data assembly, and 4) query execution. Query analysis denotes the extraction of metadata from a query. Cache access denotes total access time; tables 7 and 8 show a more detailed view on the constituent parts. Data assembly represents the time needed to combine the retrieved data triples into a single RDF graph, on which the query will be executed. Here, the total number of returned triples is also shown, illustrating the fine-grainedness of the data retrieval. Finally, we show the total amount of time required for query resolving (together with the number of query results in brackets).

Table 5. Query execution times for source-cache (ms)

query	query analysis	cache access	combine data		query execution	total
			# triples	time		
Q1	195	2463	784	81	34 (6)	2773
Q2	46	47383	5460	1541	1105 (409)	50075
Q3	32	1068	550	35	17 (4)	1152
Q4	31	28593	10076	985	37 (4)	29646
Q5	40	61635	15898	1564	515 (214)	63754

Table 6. Query execution times for meta-cache (ms)

query	query analysis	cache access	combine data		query execution	total
			# triples	time		
Q1	29	3090	6	5	7 (6)	3131
Q2	31	39789	1061	464	301 (409)	40585
Q3	30	1654	4	1	10 (4)	1695
Q4	15	17383	371	261	21 (4)	17680
Q5	57	9276	5965	2908	123 (214)	12364

Tables 7 and 8 show the times related to cache access and maintenance. Cache access comprises: 1) SIM access time, 2) time required to retrieve the cache units (the number of returned units is also shown in brackets), and 3) number of cache misses (the resulting amount of sources to be downloaded is shown in brackets), together with the retrieval times for missing cache data. Note that in case of meta-cache, the latter not only includes the download time but also the time required to extract the relevant triples. In case of source-cache, the SIM is employed to identify relevant sources (present in the cache or not); meta-cache does not employ the SIM for cache element identification. Cache maintenance comprises adding new source data and updating the cache (in case missing data was downloaded), and running the replacement strategy if space is needed. As cache maintenance occurs after query execution, it is not included in the cache access times shown in tables 5 and 6.

Table 7. Cache access / update times for source-cache (ms)

query	Cache access				Cache maintenance	
	SIM access	cache retrieval	cache miss		add / update	replacement
			# misses	retrieval		
Q1	4	2087 (5)	1 (1)	372	0	11806
Q2	127	25008 (254)	72 (72)	22248	0	31043
Q3	1	1067 (4)	0 (0)	0	0	318
Q4	7	17282 (59)	24 (24)	11304	1	24818
Q5	75	61560 (328)	0 (0)	0	0	70455

Table 8. Cache access / update times for meta-cache (ms)

query	Cache access				Cache maintenance	
	SIM access	cache retrieval	cache miss		add / update	replacement
			# misses	retrieval		
Q1	0	4 (0)	2 (6)	3086	10	0
Q2	0	1994 (6)	4 (112)	37795	64	572
Q3	0	2 (0)	1 (4)	1652	4	0
Q4	0	477 (16)	21 (40)	16906	219	193
Q5	0	9276 (6)	0 (0)	0	0	6251

5.3. Discussion

First, we discuss the results for the identification and selection of sources. Regarding space overhead (see table 1a), we observe that any of the SIM variants stores only a very small fraction of the total dataset (the largest, SIM3, stores around 0,4%), complying with the requirement for minimal storage space set in section 3. Nevertheless, it should be noted that space overhead amounts to around 2Mb for the largest SIM (SIM3) for 2500 sources. Considering Android applications have a max heap space of 16Mb (not exclusive for use by the SIM), a sufficiently larger amount of sources would require swapping parts of the SIM to persistent storage, decreasing performance. The computational overhead of extracting and adding data (see table 1b) is reasonable, complying with the requirement for minimal computational overhead. It can therefore be observed that, while the computational and storage overhead rises with the SIM complexity, this overhead appears to be acceptable (making the SIMs indeed lightweight). As expected, the selectivity of the SIM increases with the amount of metadata stored (see table 2a). The selectivity of SIM1 is so poor that an out-of-memory error occurs when assembling the sources for three of the queries (see table 2b); for no SIM (i.e. native query engine performance), an out-of-memory error already occurs for 400 sources. We thus observe that a considerable gain is made in query execution time for SIM2 and SIM3. The difference in selectivity between SIM2 and SIM3 is only visible for queries 1 and 3. This is because these queries constrain the object types of most triple patterns, allowing SIM3 to be more selective. We may thus conclude that the best SIM depends on the posed queries and is application dependent. In our cache experiments, we opted to work with SIM3, as we found the potential increase in selectivity makes up for its (relatively small) overhead.

We now elaborate on the results of caching downloaded source data. First, we consider the cache space and build times (see tables 3 and 4). The additional in-memory storage, taken up by cache index data, is about 0,04% of the total source dataset size for source-cache, while meta-cache requires 0,36%. For meta-cache, this may again lead to memory issues for larger sets, requiring swapping to persistent storage with performance loss. We also observe that, as expected, the total cost of adding new sources is higher for meta-cache, as it requires triple metadata to be extracted. Also, a single source may require updating many units in meta-cache, leading to a higher update time. On the other hand, cache replacement is less costly for meta-cache, because cache units are more fine-grained and thus replacement (i.e. moving units to persistent storage, removing units) is more effective.

With regards to query execution (see tables 5 and 6), our results show that meta-cache retrieves data in a much more fine-grained way than source-cache (i.e. less amount of triples), leading to lower overall data combination and query execution times. Looking in more detail at the cache access times (see tables 7 and 8), we observe that in order to serve a given query, source-cache also requires more cache elements to be retrieved than meta-cache. Indeed, typically a large number of sources contain data relevant to posed queries (e.g. certain predicates). This leads to much higher cache retrieval times for source-cache, and also more cache misses on average. However, although the number of cache misses for meta-cache is smaller, the actual number of downloaded sources is much higher. When a cache unit is removed and later referenced again (cache miss), all sources containing the associated metadata need to be re-downloaded. In case this metadata is contained in a large number of sources (e.g. query 2 and 4), the associated source retrieval time becomes exceedingly high. For more complex queries (requiring more cache units), this higher cache miss overhead can be compensated by the much lower cache retrieval times (query 2 and 4). Regarding cache maintenance, replacement times are again much higher for source-cache than for meta-cache. As larger, more coarse-grained cache units are retrieved and employed to serve a given query (becoming “recently used”), fitting these units into the available cache space takes more time.

To conclude, meta-cache considerably outperforms source-cache for the three more complex queries due to the smaller granularity of retrieval, and is only slightly slower for the two simpler queries. In any case, the cache maintenance overhead is much smaller for meta-cache. This improved execution and cache maintenance time outweighs the extra overhead incurred during the indexing phase. However, cache misses present a problem for meta-cache and greatly reduce performance. More advanced and fine-tuned replacement policies could be investigated to avoid cache misses, or the available cache space may be increased. Also, the architecture could be extended to reduce the amount of downloaded sources (see future work).

6. Related work

Our solution for transparent, efficient querying of a large set of small, online RDF sources is based on two pillars: indexing and caching. Below, we elaborate on both.

In other fields, indexing is a well-known technique to optimize data access. In the field of query distribution, metadata indices are employed to divide a query into subqueries and distribute them over datasources containing relevant data. Often, additional information to optimize the query distribution plan is also stored. For example, Quilitz et. al [7] use a service description containing information about found predicates, together with statistical information such as the amount of triples using a specific predicate and certain objects (e.g. starting with letters A to D). In [8], characteristics about the data provider are also kept, such as the data production rate. In [9], full-text indices are used to determine which peers contains particular triples. So-called source-index hierarchies are employed in [10], which enable the identification of query endpoints that can handle combinations of query triple patterns (or "paths"), to reduce the number of local joins. Although we share a common goal, namely identifying relevant datasources, these approaches focus on keeping index information to optimize query distribution. In the context of RDF stores, full-resource indices (i.e. indexing found s/p/o resources and potentially combinations thereof) are often employed in RDF stores, to speed up access to the RDF data (e.g. androjena, HexaStore [11], RDF On the Go [6]). However, as noted in [11] and similar to full-text indices [9], such indices are very memory and computationally intensive, with high update and insertion costs. Therefore, the index structures from these fields do not comply with the requirement discussed in section 3; namely, that a source index in a mobile setting should be lightweight to construct and update, and compact in size.

The goal of client-side caching is to exploit the capabilities of client devices, such as storage space and processing power, to increase performance and scalability of a system [12, 13]. Most existing caching approaches are based on client-server architectures, where all necessary data can be obtained from the server. In traditional data-shipping techniques, clients perform queries locally on data obtained from a server; the data can then be cached for later re-use [12]. In case of a cache miss, the missing tuples (or pages) are obtained by sending their identifiers to the server. In our setting, such caching cannot be directly applied, as no single server exists defining such unique identifiers. In query caching, query results are cached and re-used by future queries, by using query folding techniques [14]. When the cached query results are not sufficient to answer a new query, a remainder query is generated to obtain the missing data from the server in a fine-grained way. In our approach, there is no possibility to obtain specific non-cached data items. Instead, the corresponding full sources need to be downloaded, defeating the purpose of the remainder query. Comparable to query caching, we group triples in the cache according to the semantics of the cached data. However, instead of relying on posed queries to define these semantics, we exploit the inherent semantics of the cached data.

Ample work has been put in the development of cache replacement functions fine-tuned towards mobile environments. Unlike traditional replacement policies (e.g. relying on temporal locality), such functions utilize semantic locality, where general properties and relations of the data items are exploited. For instance, in [12], cached query results associated with physical locations furthest away from the location of the latest query are removed. In the FAR policy [15], cached units not located in the user's movement direction *and* furthest away from the user are removed. As our query service is not targeted to one single application type (e.g. location-aware systems), we currently rely on the generic LRU strategy to replace cache units. Future work consists of investigating alternative replacement strategies (see next section).

7. Conclusions and future work

We have presented a query service for the efficient and transparent querying of large numbers of small online sources. In order to achieve this efficient access, we rely on 1/ indexing and selection of query-relevant sources, based on semantic source metadata, and 2/ caching of often-used downloaded source data. For each component we have realized different variants, taking into account the requirements that exist in a mobile, volatile setting. For the indexing component, our evaluation has shown that significant reduction in query execution time can be reached for SIM2 and SIM3. SIMs storing more meta-data perform better due to increased selectivity, but also cause increased overhead; therefore, a trade-off needs to be made. Regarding the caching component, we found that organizing the cached data around their metadata (i.e. predicate and type information) significantly increases the fine-grainedness of cached data retrieval and overall cache performance. At the same time however, cache misses present a serious overhead for this kind of cache organization.

Future work consists of minimizing the effects of cache misses, by exploring more advanced replacement policies (e.g. location-aware) and investigating source-level replacement in meta-cache. Also, we aim to investigate the effectiveness of replacement policies in different mobile scenarios (potentially selecting suitable ones automatically). Finally, to deal with the limited storage of mobile devices, two-level indices that can be efficiently swapped to persistent storage should be investigated.

Acknowledgement

Sven Casteleyn is supported by an EC Marie Curie Intra-European Fellowship (IEF) for Career Development, FP7-PEOPLE-2009-IEF, N° 254383.

References

[1] P. Bellavista, A. Corradi, R. Montanari, A. Toninelli, Context-Aware Semantic Discovery for Next Generation Mobile Systems, in: Communications Magazine, IEEE. 44(9), 62-71, 2006.

[2] V. Reynolds, M. Hausenblas, A. Polleres, Exploiting Linked Open Data for Mobile Augmented Reality, in: Proc. of W3C Workshop: Augmented Reality On the Web, Spain, 2010.

[3] S. Casteleyn, W.V. Woensel, O.D. Troyer, Assisting Mobile Web Users: Client-Side Injection of Context-Sensitive Cues into Websites, in: Proc. of 12th International Conference on Information Integration and Web-based Applications & Services, 443-450, France, 2010.

[4] W.V. Woensel, S. Casteleyn, O.D. Troyer, A Framework for Decentralized, Context-Aware Mobile Applications Using Semantic Web technology, in: Proc. On the Move to Meaningful Internet Systems: Workshops, 88-97, 2009.

[5] Melinger, D., Bonna, K., Sharon, M., SantRam, M. Socialight: A Mobile Social Networking System. Proc. of the 6th International Conference on Ubiquitous Computing, 429 - 436, England, 2004.

[6] D. Le-phuoc, J.X. Parreira, V. Reynolds, RDF On the Go: An RDF Storage and Query Processor for Mobile Devices, in: Proc. of 9th International Semantic Web Conference (ISWC2010), China, 2010.

[7] B. Quilitz, U. Leser, Querying Distributed RDF Data Sources with SPARQL, in: Proc. of 5th European Semantic Web Conference, Spain, 2008.

[8] S. Lynden, I. Kojima, A. Matono, Y. Tanimura, Adaptive Integration of Distributed Semantic Web Data, in: Databases in Networked Information Systems 5999, 174-193, 2010.

[9] Z. Kaoudi, K. Kyzirakos, M. Koubarakis, SPARQL Query Optimization on Top of DHTs, in: Proc. of 9th Int. Semantic Web Conference, China, 418-435, 2010.

[10]H., Stuckenschmidt, R., Vdovjak, G.J. Houben, J. Broekstra, Towards Distributed Processing of RDF Path Queries, in: International Journal of Web Engineering and Technology 2(2/3), 207-230, 2005.

[11] C. Weiss, A. Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, in: Proc. of VLDB Endowment 1(1), 1008-1019, 2008.

[12] Dar, S., Franklin, M. J., Jónsson, B., Srivastava, D., & Tan, M. Semantic Data Caching and Replacement. Proc. of the 22th Int. Conference on Very Large Data Bases, USA, 330-341, 1996.

[13] Jónsson, B., Arinbjarnar, M., Bórsson, B., Franklin, M. J., & Srivastava, D. (2006). Performance and overhead of semantic cache management. *ACM Trans. Int. Technology*, *6*(3), 302-331, USA, 2006.

[14] Ren, Q., Dunham, M. H., & Kumar, V. Semantic Caching and Query Processing. *IEEE Trans. on Knowl. and Data Eng.*, *15*(1), 192-210. P, USA, 2003

[15] Ren, Q., & Dunham, M. H. Using semantic caching to manage location dependent data in mobile computing. *MobiCom '00: Proc. of the 6th annual int. conference on Mobile computing and networking*, 210-221, USA., 2000.