A Mobile Query Service for Integrated Access to Large Numbers of Online Semantic Web Data Sources

William Van Woensel (*corresponding author*)[1]
[1] Dalhousie University
6050 University Avenue, Halifax NS B3H 4R2, Canada
William.Van.Woensel@Dal.Ca

Sven Casteleyn[2, 3]
[2] Universidad Jaime I
E-12006, Castellón de la Plana, Spain
[3] Vrije Universiteit Brussel
B-1050, Brussels, Belgium
Sven.Casteleyn@uji.es

From the Semantic Web's inception, a number of concurrent initiatives have given rise to multiple segments: large semantic datasets, exposed by query endpoints; online Semantic Web documents, in the form of RDF files; and semantically annotated web content (e.g., using RDFa), semantic sources in their own right. In various mobile application scenarios, online semantic data has proven to be useful. While query endpoints are most commonly exploited, they are mainly useful to expose large semantic datasets. Alternatively, mobile RDF stores are utilized to query local semantic data, but this requires the design-time identification and replication of relevant data. Instead, we present a mobile query service that supports on-the-fly and integrated querying of semantic data, originating from a largely unused portion of the Semantic Web, comprising online RDF files and semantics embedded in annotated webpages. To that end, our solution performs dynamic identification, retrieval and caching of query-relevant semantic data. We explore several data identification and caching alternatives, and investigate the utility of source metadata in optimizing these tasks. Further, we introduce a novel cache replacement strategy, fine-tuned to the described query dataset, and include explicit support for the Open World Assumption. An extensive experimental validation evaluates the query service and its alternative components.

Keywords: mobile computing; data integration; data indexing; data caching; cache replacement; open world assumption

1. Introduction

The Semantic Web has grown with leaps and bounds over the last decade. Large data sources have been put online in semantic format, and made interoperable via initiatives such as Linked Data [1] (e.g., DBPedia, LinkedGeoData). In addition, small online RDF files, for instance capturing item descriptions (e.g., using DCMI) or personal profiles (e.g., using FOAF), also constitute a large part of the Semantic Web. Sindice [2], a Semantic Web search engine, indexes ca. 708 million of these online sources. In a parallel evolution, increased efforts are being made to make regular (HTML) web content machine-readable as well, catalyzed by the commitment of major search engines to leverage such annotations for improving search results [3]. This evolution has given rise to a new Semantic Web segment, comprising web content enhanced with semantic annotations (e.g., RDFa, microdata). Since most of this annotated content can be converted to RDF data (e.g., see [4]), such annotated websites are semantic sources in their own right. The Web Data Commons initiative [5] (2013) found that ca. 26% of crawled webpages already contain semantic annotations.

Via the Semantic Web, mobile clients gain access to a wealth of online, freely available knowledge. Various mobile computing domains currently leverage semantic data, including augmented reality [6, 7], recommender systems [8], location-aware [9, 10] and context-aware systems [11, 12], mobile tourism [13] and m-Health [14]. Typically, these systems access online semantic data via SPARQL query endpoints. Since they relieve mobile clients of computationally intensive query resolution, query endpoints represent an efficient option for mobile clients. On the other hand, client-server roundtrips cause delays, and a poor or unavailable network connection prevents query resolution. Furthermore, setup and maintenance incur costs, especially when scalability is desired, and requires technical expertise and effort. Therefore, they only present an acceptable cost-benefit ratio for large RDF datasets.

Given recent improvements in mobile hardware, coupled with the development of mobile query engines, an alternative is the local querying of semantic web data [12, 15, 16]. However, local querying requires the manual, a priori replication of relevant data, and gives rise to data freshness issues. Moreover, some domains do not allow establishing data relevance beforehand; e.g., in context-awareness, relevance is determined by the mobile user's current context, which is updated continuously and often in unforeseeable ways. Due to nearly ubiquitous wireless

connectivity, opportunities currently exist to bypass these drawbacks and dynamically retrieve relevant semantic data.

We present a client-side, general-purpose mobile query service, to study the performance and feasibility of on-the-fly querying of a mainly untapped portion of the Semantic Web, consisting of large amounts of RDF files and annotated websites. By supplying integrated query access over these sources, the query service can resolve distributed queries, referencing data from multiple sources. In particular, our solution relies on the dynamic identification, retrieval and caching of semantic data relevant to posed queries. For this purpose, the query service includes two key components; 1/ a source identification component, to identify query-relevant sources in the online semantic dataset, and 2/ a cache component, locally storing data for later re-use. The query service relies on an existing mobile query engine to locally query retrieved RDF data. To reconcile fine-grained data selection with reducing data processing overhead, these components exploit the semantics of RDF(S)/OWL data.

Studying the efficiency and workability of such local, client-side data collection and query support is desirable for a variety of reasons. First, it is an infrastructure-less solution, where no single party needs to invest in highly scalable server infrastructure or cloud subscriptions. Secondly, keeping data and posed queries at client-side ensures privacy, e.g., especially in context-aware scenarios. Third, by collecting data locally, it ensures query capability for applications in conditions of poor or unreliable network connection. Even with sufficient Internet connectivity, local querying avoids client-server roundtrips, which potentially decrease performance at query time, which is most critical. Finally, it very well supports application scenarios where semantic data fragments are retrieved by other means than the Internet (e.g., via Bluetooth connection, from high capacity RFID tags).

This article builds on earlier work [17], where we presented preliminary versions of the main query service components. In this article, we present an elaborated version of the query service, including extensions that tackle previously identified shortcomings. These include a novel cache removal strategy called Least-Popular-Sources (LPS), tailored to our particular situation where cached data originates from online data files. Secondly, in order to fully support integrated Semantic Web querying, we incorporated the Semantic Web Open World Assumption (OWA). Our experimental validation evaluates the query service using a larger, real-world dataset, focusing on the effects of these extensions on performance and completeness of query results; while at the same time studying boundaries of semantic web technology on current mobile devices.

In the remainder of this article, we first discuss challenges and requirements that arise in our particular querying scenario, together with suitable solutions. Next, an overview of the query service is presented, and its general phases are discussed. We continue by detailing the major query service components, as well as the LPS strategy, and further discuss built-in support for the Semantic Web OWA. Subsequently, the query service is evaluated via an experimental validation. We proceed with a review of the state of the art and end with conclusions and future work.

2. Challenges and requirements

The goal of our mobile query service is to provide transparent, integrated access to a currently untapped part of the Semantic Web, comprising online RDF files and annotated websites. In this mobile querying scenario, a number of issues and challenges arise, which we discuss below.

1. Mobile device restrictions: although mobile devices are catching up with desktop and laptop computers, they still have limitations regarding processing and memory capacity (e.g., Android applies a maximum heap depending on the device; currently, for devices with 2 – 3 gigabyte of RAM, this limit is typically 128-192MB per Android 5.1 app). Furthermore, battery power is limited, and restricts full and continuous utilization of hardware resources.

2. Large query dataset: due to its scale, it is impossible to consider the entire Semantic Web as query dataset. Reflecting this, existing approaches to integrated querying only focus on a (configured) Semantic Web subset. However, our experiments (see Section 7) show that querying even moderately sized datasets is currently not feasible on mobile platforms (e.g., the entire dataset needs to be kept in-memory for fast querying).

3. Dynamic Semantic Web subset and volatile semantic sources: typically, mobile applications only require access to a specific Semantic Web subset; ruling out the need to consider the entire Semantic Web (see above). For instance, context-provisioning systems [17] require access to semantic context sources (e.g., place descriptions); while recommender systems [8] require semantic descriptions of items to be recommended. Often, these datasets are only known at runtime and subject to change, which

necessitates allowing mobile apps to delineate and dynamically extend their relevant Semantic Web selection [18]. Furthermore, semantic sources themselves may change over time. Depending on the usage scenario, changes may be only occasional (e.g., product descriptions in e-commerce) or frequent (e.g., semantic Internet of Things). In any case, our query service needs to be able to cope with a dynamic set of potentially evolving sources.

4. Data captured in online, third-party files: in our querying scenario, data items originate from third-party online files. To gain access to their comprised relevant data, such files need to be fully downloaded, thus retrieving both relevant and irrelevant data. As such, data-retrieval overhead is significantly increased. We also note that connectivity interruptions, not uncommon in mobile scenarios, will result in the query dataset becoming inaccessible.

Taking into account these observed challenges, we formulate the following requirements for efficiently querying large sets of online semantic sources:

1. *Minimizing resource usage*: a local query service should not strain mobile memory and processing capacities, nor overly drain the device's battery (challenge 1). Since only a relatively limited amount of fast, volatile memory is available (challenge 1), any additional (volatile) memory requirements (e.g., to store supporting index structures) need to be minimal. Ideally, the additional data should fit in volatile memory to avoid frequent swapping with persistent storage, which unavoidably causes performance loss. Secondly, as mentioned, the query service should enable mobile applications to delineate and dynamically extend or update their relevant Semantic Web selection (challenge 3). This means any internal data structures need to be updateable in real-time and with minimal computational effort, while still supporting acceptable query performance. Finally, battery consumption should be kept within acceptable bounds. For instance, this means reducing battery-intensive operations as much as possible, such as source downloads, which require WiFi or 3/4G radios, and large-scale persistent data retrieval.

2. Minimizing query dataset: querying large datasets causes performance problems, especially on mobile platforms (challenge 2). Barring extraordinary mobile hardware improvements in the near future, this implies the query dataset should be kept as small as possible, while still allowing complete query results to be returned. Reducing the query dataset is also tackled in other related approaches, such as query distribution [19, 20] and context information systems [12, 21]. For instance, query distribution systems typically focus on ruling out datasets irrelevant to posed queries.

3. Minimizing online data downloads: retrieving online query data is inherently expensive, both in time and battery use (challenge 1), and constrained by connectivity (challenge 4). Lack of control over online data files prevents more efficient solutions at the source side, such as selectively downloading only relevant parts, or only re-downloading updated parts in case of evolving data sources (challenge 3). As such, data retrieval should be avoided where possible. For instance, this can already be (partially) achieved by reducing the query dataset (req. 2) and thus the number of sources to (re-)download.

In order to meet these requirements, two solutions present themselves:

- *Fine-grained identification of relevant sources:* by identifying data relevant to application queries in a fine-grained way, the query dataset can be greatly reduced (req. 2), as well as the number of relevant sources to download (req. 3). Such identification may occur pro-actively, before any queries have been posed; or re-actively, for each individual posed query. For instance, domain-specific approaches exist [12, 21] that pro-actively and dynamically locate useful Semantic Web data, in this case by correlating the information to the user's context. Since pro-active data identification is not always possible (e.g., in case relevance is determined by user input), it is not a suitable choice for our general-purpose query service. Therefore, we choose a re-active approach, comparable to query distribution approaches [19, 20]. As an important advantage, this approach directly supports any scenario encapsulated by application queries (e.g., context-awareness, recommendation). However, it also requires identification and source retrieval tasks to occur during query resolution, increasing total resolution times. In any case, efficient identification can be supported by indexing source data on-the-fly, as the mobile application delineates its relevant Semantic Web subset. As indicated by req. 3, download overhead can already be mitigated by fine-grained data selection; as well as by applying the second solution, *Locally caching data*.

- *Locally caching data*: by locally caching online data, fewer sources need to be (re-)downloaded to serve a posed query (req. 3). Using caching, the query resolution time is decreased by avoiding source re-downloads; thus reducing the drawbacks introduced by pro-active source selection (see first solution). By

further allowing cached data to be retrieved with high-selectivity, the query dataset can be further reduced (req. 2). When applying caching, storage footprints are kept in check by applying replacement policies (a.k.a. removal strategies). In mobile settings, the need for caching is reflected in related work [22, 23]. To avoid cache invalidity caused by evolving sources, a flexible cache validation strategy needs to be deployed, which accommodate datasets evolving at different rates and avoids unnecessary data (re-)downloads.

Given our first requirement of minimizing resource usage (req. 1), our main goal is to find a good balance between the proposed *fine-grained data retrieval*, afforded by effective data indexing and local caching; and the *memory and computational overhead* this implies, e.g., resulting from supporting data structures. As only a well-balanced solution will provide good query resolution times, our research seeks to harmonize these counteracting concerns.

3. General approach

The query service implements the two proposed solutions, namely identifying relevant online sources and locally caching data, via two key components. Importantly, both components rely on source metadata, which includes found predicates and resource types, to achieve their task. The *source identification component*, called the **Source Index Model (SIM)**, indexes online source metadata from online semantic sources, with the goal of enabling fine-grained source identification. The *cache component* locally caches downloaded source data and has two variant implementations, called **Source Cache** and **Meta Cache**. Each variant presents a different cache organization: Source Cache organizes cached data around origin source, while Meta Cache arranges the data based on shared source metadata. Multiple SIM variants were developed as well, each keeping increasing amounts of metadata. By developing multiple component variants, we aim to study the utility of the aforementioned metadata in achieving our goal; namely, reconciling fine-grained data retrieval with reduced memory and processing overhead (see Section 2). Below, we discuss the rationale behind our focus on source metadata.

Source metadata, including predicates and resource types, can be easily and efficiently retrieved from semantic sources. Compared to instance-level information, as indexed by RDF stores or certain query-distribution approaches (see related work, Section 8), extracting this metadata is less processing-intensive; while much less data needs to be indexed as well, decreasing memory usage. At the same time, we hypothesize that source metadata still allows for fine-grained data retrieval, which is confirmed by our experiments (see Section 7).

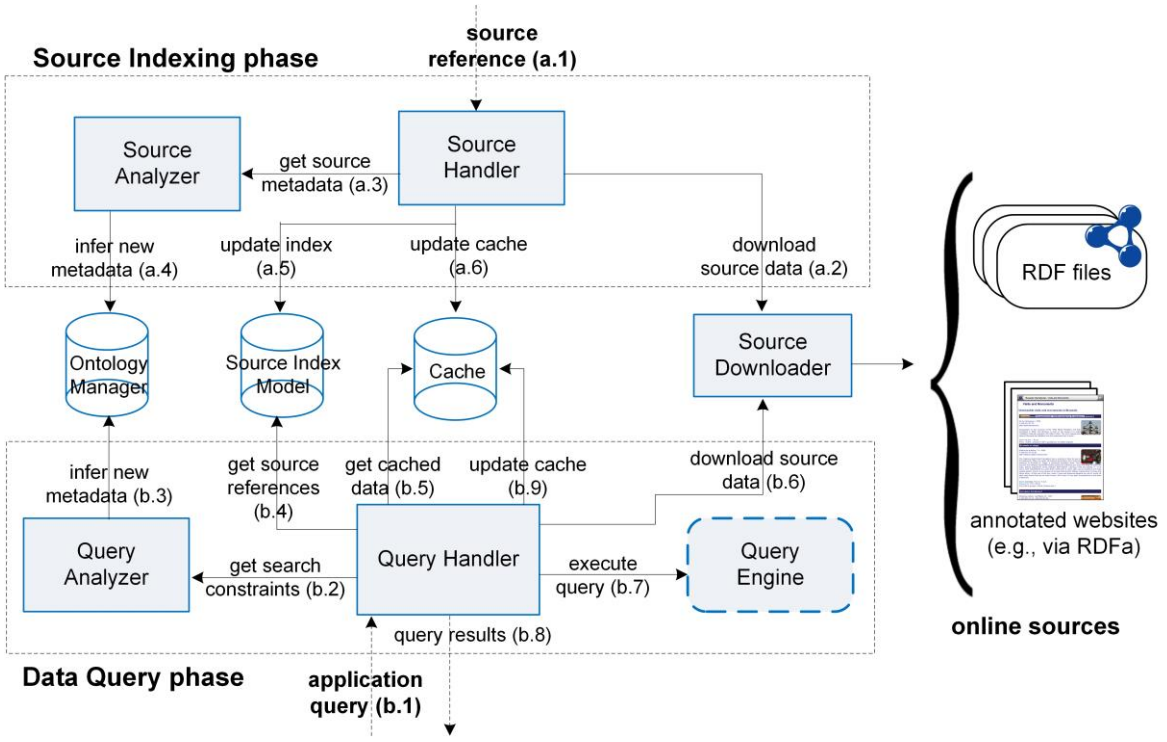

Figure 1. Overview of the components and phases of the mobile query service.

Clearly, before it can be utilized for source identification and caching, such source metadata needs to be present in 1/ online sources and 2/ posed queries. The real-world dataset gathered for our experimental evaluation, extracted from a range of existing online sources, confirms that online sources typically specify subject/object types to describe contained resources. Furthermore, semantic queries often specify concrete predicates and constrain subject/object types of query variables. Both these observations are reflected in the related domain of semantic query distribution, where approaches index RDF predicates [19, 24] and types [20] to identify query-relevant datasets.

However, indexing any kind of RDF data inevitably raises problems resulting from the Semantic Web's Open World Assumption (OWA) and its inherently distributed nature. Due to its lack of negation-as-failure, the OWA implies that no single source is self-contained or complete; other sources can thus specify additional information for each resource. In our case, this means multiple different resource types can be specified across different, distributed online sources, potentially leading to inconsistent indexed metadata. We elaborate on this issue in Section 6.

Figure 1 shows an overview of the mobile query service components and phases. The query service relies on an existing mobile query engine (e.g., AndroJena [25], RDF On The Go [26]) to locally query the downloaded semantic data. Below, we discuss each phase in more detail.

The Source Indexing phase is triggered when the client (i.e., mobile app utilizing the query service) passes the location of an a priori known or newly discovered online source (a.1), allowing the app to outline its relevant portion of the Semantic Web. For applications where the required dataset is known beforehand, this may occur in bulk [8] whereby updates may be issued later on; in other cases, this will happen gradually and in real-time [17]. In our experimental evaluation (see Section 7), the SCOUT mobile context-provisioning framework [11] acts as client, passing online sources describing the user's physical environment as they are discovered. Other client apps can also be envisioned, including any type of context- or environment-aware application (e.g., mobile tourism applications, such as restaurant finders, museum guides, city tour apps, etc.; m-commerce application, such as geo-fenced coupon apps, shopping comparison apps, real-estate apps, etc.), and other applications scenarios (e.g., mobile recommender systems, such as music or movie recommenders; aggregator apps, such as news or search aggregators; or social-networking-based applications, such as dating or travel apps). Depending on the concrete scenario, the delineation and dynamic expansion of the relevant dataset may also come in different forms: discoverable (i.e., in context-aware scenarios), computable (i.e., as a result of an on the fly crawling process), or previously known.

Upon receiving an online source reference (a.1), the Source Handler contacts the Source Downloader to retrieve the source data (a.2). In addition to online RDF files, the Source Downloader also supports semantically annotated websites, automatically extracting their annotations as RDF triples (currently, RDFa is supported). The retrieved source data is then passed to the Source Analyzer (a.3), which extracts the required *source metadata*, including predicates and resource types. The Source Analyzer can optionally employ the Ontology Manager to infer additional metadata, based on axioms from well-known ontologies (a.4). After extraction, the source metadata is passed to the Source Index Model (SIM) for indexing (a.5), and the downloaded source data is passed to the cache component for storage (a.6).

The Data Query phase commences when the client poses a query (b.1). The given query is first analyzed by the Query Analyzer (b.2), which extracts query metadata as *search constraints*. This query metadata reflects the extracted source metadata, and comprises concrete predicates and type constraints. As before, the Query Analyzer may utilize the Ontology Manager to infer additional query metadata (b.3). The Query Handler then passes the extracted search constraints to the SIM, which returns references to online sources containing relevant data (b.4). Given the identified source references (and extracted search constraints[1]), the cache component is contacted (b.5), returning query-relevant source data locally available in the cache. Any sources not found in the cache, due to applied removal strategies (in case storage was full), are re-downloaded by the Source Downloader (b.6).

Afterwards, an existing mobile query engine executes the query (b.7) over the collected query dataset, after which the query results are returned to the client (b.8). At the end of the phase, the cache is updated with the (re-)downloaded source data (b.9).

As mentioned, the Source Indexing and Data Query phases are respectively triggered when indexing an online source and executing an application query. In

[1] Meta Cache relies on the search constraints for retrieving cached data (see Section 5.1.2).

case new sources are discovered dynamically, they are thus likely to occur intermittently at runtime. The query service is implemented for the Android platform (version 4.1.2). The AndroJena library supplies the mobile query engine, though any other mobile query engine can be used.

Below, we elaborate on the concrete implementations of our solutions, namely identifying online sources and caching source data.

4. Identifying relevant online sources

By indexing online source data, query-relevant sources can be identified during querying. In particular, the Source Index Model (SIM) focuses on source metadata, including predicates and resource types, resulting in a compact index that is quick to update and maintain, while still ensuring high source selectivity. Given analogous metadata extracted from queries, the SIM utilizes the indexed metadata to identify query-relevant sources in a fine-grained way.

To validate the effectiveness of source metadata in reconciling data selectivity and overhead, we developed 3 SIM variants, each keeping increasing amounts of metadata: SIM1, only storing predicates, SIM2, keeping predicates and subject types, and SIM3, keeping predicates, subject and object types.

Below, we shortly elaborate on the index structure employed by the SIM. Then, we discuss the source and query analysis and source identification processes.

4.1 Source Index Model

The Source Index Model is implemented using a multi-level index; a type of index used traditionally in databases, but in this case specifically tailored for source identification based on source meta-data. In the related work section (see Section 8), we discuss other indexing structures employed by related approaches. Each index level indexes on a particular metadata part (i.e., predicates, subject or object type), and keeps maps that connect metadata parts occuring together in source triples. In particular, the first-level map indexes on predicates, whereby each entry links to a second-level map keeping subject types. Each subject type further links to a map keeping object types, each of which finally points to a list of URLs. Each linked combination of predicate, subject and object type (i.e., path through the multi-level index) indicates that the particular metadata combination occur together in one or more triples from the indicated sources. Given that sources may contain triples without types, an <empty> map entry may be added as well. For instance, a predicate entry linking to an <empty> subject type and <empty> object type entry indicates the predicate was found without subject/object types in the indicated sources.

To reduce the size of the SIM, dictionary encoding is applied (similar to RDF stores [27, 28]). This encoding process is fine-tuned towards RDF terms, and maps namespaces (indicating a set of related resources) to an integer identifier, while local names (indicating the concept or item) are kept as character arrays. We found this resulted in the largest size reduction, as namespaces are repeated across data sources much more often than the local names.

4.2 Source Analysis

The Source Analyzer extracts metadata for each retrieved online source, including predicates, subject and object types. Initially, this extraction was realized via predefined SPARQL extraction queries [17, 29]. However, this led to huge processing overheads when dealing with real-world sources, which contained large amounts of distinct metadata. We therefore optimized the metadata extraction process by dynamically parsing RDF files in N-TRIPLE format (which are straightforward to parse), processing the RDF line-per-line and returning new RDF metadata statements as requested by the Source Analyzer. This way, we avoid an expensive RDF graph creation (performance and memory-wise) and querying step. This resulted in an average performance gain of factor 10.

4.3 Query Analysis

The Query Analyzer analyzes each triple pattern in a query's WHERE, OPTIONAL and UNION clauses to retrieve query metadata, including predicates and resource types, which can then be matched to indexed source metadata. FILTER clauses are further scanned for functions indicating equivalence between variables and resources (i.e., `sameTerm` function), which may result in additional concrete predicates and types.

```
SELECT ?place
WHERE {
   ?person rdf:type foaf:Person .
   ?person foaf:based_near ?place .
   ?place rdf:type rest:Restaurant .
}
```

Code Listing 1. Example SPARQL query and extracted triple patterns

Code Listing 1 shows an example SPARQL query, where the underlined triple patterns supply type constraints for the triple pattern in bold. The following query metadata combination, or search constraint, is extracted for Code Listing 1: *foaf:based_near – foaf:Person – rest:Restaurant*.

The Query Analyzer utilizes the SPARQL Parser library [30] to parse SPARQL queries, and then visits the parsed Abstract Syntax Tree (AST) to extract the search constraints.

4.3 Source Identification Process

To identify query-relevant sources, search contraints extracted from queries are matched with metadata from online sources. In particular, the SIM follows each individual search constraint as a path through the multi-level index. Respectively using the predicate, subject and object type as keys, the predicate index returns a subject type index (predicate key), which in turn leads to an object type index (subject type key). Finally, this latter index returns a list of source URLs (object type key), each adhering to the given search constraint. By performing this step for each separate search constraint, as opposed to the entire query, sources can be identified for queries that are not solvable by any single source, but require a combination of sources; thus supplying full integrated query access across online sources.

In case subject/object variables of a triple pattern have multiple type restrictions (e.g., `foaf:Person`, `dcmi:Agent`), a data source is only relevant for the triple pattern if it specifies all given types for that variable. To realize this, separate search constraints are extracted for each type, and an index search is performed for each constraint. Afterwards, the intersection of the found sources is taken, ensuring the sources each adhere to the extracted constraints.

Similar to sources, some queries may lack certain metadata, including types and predicates. In this case, missing metadata indicates no constraint is given on the missing metadata part(s) (e.g., subject type). This means the search at the particular index level (e.g., subject type index) is unconstrained, and all entries at the particular index level need to be followed (e.g., subject type index). Afterwards, the union of all found sources is taken, denoting all sources that fulfil the (partial) constraint. Note that the <empty> entry, indicating a lack of particular source metadata (e.g., subject type), only matches if the search is unconstrained at that level.

By considering each search constraint separately, as well as supporting both missing source and query metadata, all sources containing query-relevant data are returned. However, full completeness can only be guaranteed if the Open World Assumption is also considered. We revisit this issue in Section 6.

5. Caching source data

Locally caching source data serves to reduce the number of source (re-)downloads required to serve a posed query. Importantly, cached data should be retrieved with high selectivity to keep the query dataset small, while additional data structures (e.g., indices) should only take up limited memory space and be quick to update and maintain. To study the extent to which source metadata can balance these two concerns, we consider multiple component variants: Source Cache, which arranges the cache according to origin source; and Meta Cache, organizing cached data according to shared metadata. In Section 5.1, we elaborate on both cache organizations, and weigh their respective advantages and drawbacks.

To manage the occupied memory and storage space, replacement policies (or removal strategies) identify data to be moved to persistent storage or removed entirely. We discuss suitable removal strategies, and detail a novel removal strategy called Least-Popular-Sources, in Section 5.2. Finally, a cache validity strategy is applied to ensure the freshness of the cache (Section 5.3). Both removal and cache validity strategies are tailored to our particular setting, where cached data originates from online data files.

5.1 Cache organizations

A cache can be organized in different ways, influencing the fine-graininess of cached data retrieval, as well as the maintenance costs and memory overhead. Cached data is indexed, stored and retrieved per unit of data called the *cache unit*, whereby the content of the unit depends on the particular cache organization.

5.1.1 Source Cache

In Source Cache, an individual cache unit contains all data from a particular online source; in other words, data is indexed, stored and retrieved per origin source. This is a natural organization in our setting, where data originates from small online sources. A search index (implemented as a hash table) is kept on source URLs, each of which uniquely identifies a cache unit. To obtain the URLs of cached, query-relevant sources, the Source Cache is deployed in combination with the SIM (see Section 4).

Since only one index is kept with a relatively small amount of entries, this cache organization results in only minimal memory overhead, while the SIM memory impact is limited as well. Cache creation and updating is also efficient, since each downloaded source is directly stored as a cache unit. On the other

hand, Source Cache does not support fine-grained data retrieval, since a retrieved cache unit comprises the entire source instead of only its relevant triples. Our experimental evaluation (Section 7) shows that this leads to high cache retrieval overheads during query resolution. As indicated by req. 3, *Minimizing online data downloads*, course-grained retrieval is unavoidable when dealing with online sources. However, this can be improved upon when dealing with local data, as shown by Meta Cache.

5.1.2 Meta Cache

In the Meta Cache organization, a single cache unit comprises all triples sharing the same metadata combination (i.e., predicate and subject, object type), irrespective or their origin source. By keeping search indices on predicates, subject and object types, relevant cache units can be quickly identified, given a particular query metadata combination.

In this case, a retrieved cache unit comprises only triples matching the query's search constraints, resulting in much more fine-grained retrieval. However, this comes with additional memory and processing overhead. Firstly, the cache update time is increased, since metadata from each source triple needs to be extracted, and added to three separate indices. Secondly, storing triples from a single online source potentially requires creating or updating multiple cache units, depending on their metadata. Regarding memory usage, three indices (implemented using hashtables) are kept with considerably more entries compared to Source Cache, since the number of distinct predicates and types usually exceeds the number of source URLs. To enable validity checking, the origin URL of each cached triple also needs to be kept (see Section 5.3). In an effort to reduce memory and storage space, type statements (i.e., with predicate `rdf:type`) are not stored, but automatically generated based on the metadata associated with retrieved cache units[2], and then inserted in the final query dataset. At the same time, we note that due to its focus on schema-level information, Meta Cache still has a much lower memory and update overhead compared to other indexing approaches (see Section 8). Additionally, our experimental evaluation shows that this overhead is still reasonable, especially when considering the resulting improvement in query resolution performance.

In addition, Meta Cache keeps information on "missing" cached data, previously removed by cache removal strategies (see next section). In particular, it keeps the metadata combination associated with the removed data, together with references to their origin sources; and indexes this information using the aforementioned indices. Consequently, a single cache lookup may return relevant cached data as well as references to online sources that need to be re-downloaded. By integrating this functionality in the Meta Cache, we rule out the need for a separate source identification component, avoiding its associated overhead. As a result, the Meta Cache implements both *online source identification* and *local caching*.

Finally, we note that, analogous to the SIM, both Source and Meta cache apply dictionary encoding to reduce memory and storage space.

5.2 Removal Strategy

In case of limited volatile and persistent storage, a removal strategy (or replacement policy) is applied to identify data to be moved from volatile to persistent storage or removed entirely, whenever volatile or persistent memory becomes full, respectively. For this purpose, well-known strategies such as Least-Recently-Used (LRU) or Least-Frequently-Used (LFU) may be employed. A number of domain-specific removal strategies exist as well, which are discussed in our related work section (see Section 8).

However, such existing strategies have the potential to cause major performance issues for Meta Cache. This is a result of the specific organization of Meta Cache, which groups source data based on shared metadata instead of origin source. As a result, cache units likely contain data originating from multiple sources. Whenever a removed cache unit is referenced during query resolution (i.e., a *cache miss*), this means all sources containing the missing metadata combination need to be fully re-downloaded, and the relevant data items extracted. This issue has its roots in our particular setting, where data is captured in online data files (see Section 2), and will have negative effects for any cache organization different from origin source. Previously, we found that this incurs a serious performance overhead during query resolution [17].

To allow for efficient query resolution when utilizing Meta Cache, we need to reduce the occurrence of this problem. For this purpose, we present a novel cache removal strategy called Least Popular Sources (LPS), which we discuss below.

5.2.1 Least-Popular-Sources

[2] E.g., for each triple "X Y Z." in cache unit with metadata <*pred1*, *subjType1*>, the type statement "X rdf:type subjType1" is generated.

Instead of removing single cache units, the LPS strategy removes all data originating from a particular source, potentially across cache units[3]. By removing data on a per-source level, cache misses resulting from a single removal only require a single source to be re-downloaded, instead of multiple sources. On the other hand, the probability of cache misses increases as well, as one source removal influences all cache units with the source's data. This is illustrated in Figure 2; by removing source A, any cache miss only incurs one source re-download; although there is now a 3/4 chance that accessing a cache unit incurs a cache miss.

Consequently, the goal of LPS is to balance 1/ the number of source re-downloads and 2/ the probability of cache misses. To that end, LPS considers the "popularity" of cached sources when identifying sources to be stored persistently or removed. As explained below, both the popularity of its source data and metadata is considered.

The first factor, *source-data popularity*, refers to the degree to which the source's data is spread across the cache, indicated by the number of cache units containing the source's data (i.e., the source data's "popularity"). As such, it marks the amount of cache units that will be affected by removing the source's data. By reducing the amount of cache units with missing data, we can decrease the probability of cache misses later on. In Figure 2, source A has the highest value for this factor, since its data is spread across 3 cache units.

The second factor, *source-metadata popularity*, reflects the number of other online sources that contain the source's metadata (i.e., the source metadata's "popularity"). Since cache units group data sharing the same metadata, origin sources participating in the same cache unit share (at least) this metadata. In case a cache unit has many origin sources, it will thus contribute to a high extent to the *source-metadata* popularity of each associated source. Applying this factor reduces the chance that many of these sources will be removed; thereby decreasing the potential number of source re-downloads on a cache miss. This is illustrated in Figure 2, where sources B, C, D and E each have three other sources keeping the same metadata (indicated by their participation in cache unit 1). As a result, these sources have a large value for this factor, reducing the likelihood that many of them will be removed. This means that a cache miss, resulting from accessing cache unit 1, will lead to only a minimal number of source re-downloads.

[3] Other sources' data in these cache units is hereby retained.

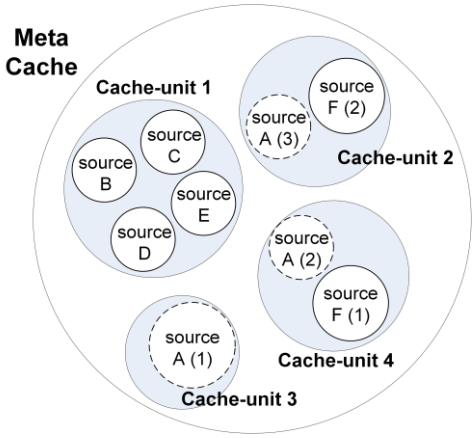

Figure 2. Example application of LPS.

In practice, these two factors allow us to cope with sources of different sizes. Small sources will typically be spread across less cache units (as they typically contain less distinct combinations of metadata), and thus have a smaller value for the *source-data popularity* factor. Due to their smaller size, a comparably large number of them also needs to be removed to clear the same amount of storage space (compared to when removing a larger source). For instance, in Figure 2, clearing storage space could involve removing the (small) sources C, D and E, resulting in 3 source re-downloads when accessing cache unit 1. However, since this cache unit contains a large amount of sources, the sources' associated value for the *source-metadata* factor is larger as well. This reduces the likelihood that many of these sources will be removed; thus decreasing the probability of many source re-downloads on a cache miss.

As a final factor, LPS can take the source's *download cost* into account, whereby sources that have long download times are less likely to be removed. Formula 1 shows the removal value calculation for source *s*, where *f1* stands for source-data popularity, *f2* for source-metadata popularity, and *f3* for download cost (in seconds)[4]. Due to the nature of *f1* and *f2*, this calculation is performed each time cache units are created, updated or removed. Different factor weights may be set, respectively represented by α, β and γ. In our experimental section, we tested different weights to find an optimal balance between these factors in our dataset (see Section 7).

$$LPS(s) = \alpha f1 + \beta f2 + \gamma f3$$

Formula 1. LSP removal value calculation.

We note that LPS was specifically designed to cope with the difficulties of cache removal in settings

[4] A higher result value means the source is less likely to be removed.

where data originates from online files. As such, it does not consider any particular locality of reference, as is typically the case for removal strategies (e.g., LRU, or Furthest-Away-Removal (FAR) [23]). Since our query service is general-purpose, it is also not possible to make a priori assumptions on likeliness of referral. Finally, we also note that LPS makes removals more complex and costly, compared to regular removal strategies. In our experimental evaluation (see Section 7), we investigate how these overheads weigh against the potential advantages.

In the section below, we elaborate on implications of LPS on cache architecture.

5.2.2 Decoupling retrieval, storage, removal units

Until now, we indicated that cached data is retrieved, stored and removed per *cache unit* (see Section 5.1). To support removal strategies such as LPS, where data is removed via a different unit (e.g., origin source), we need to further distinguish between a *retrieval*, *removal* and *storage* unit. A *retrieval unit* keeps (pointers to) the data retrieved when accessing the cache, while a *removal unit* keeps (pointers to) the data that is removed or persistently stored due to a memory management operation. A *storage unit* contains the actual cached data (in-memory/ persistent), to which retrieval and removal units point. In other words, retrieval and storage units are indexes over the actual data, stored as storage units. This allows both efficient retrieval of query data, as well as efficient removal due to cache maintenance.

In Meta Cache, the retrieval unit points to all data sharing the same metadata, and thus corresponds to the original notion of a "cache unit". When applying the LPS strategy, the removal unit will point to all data originating from a particular source. To accommodate this, the storage unit needs to be more fine-grained, keeping data *from a particular source* that *share the same metadata*. This allows removal units to keep pointers to units only keeping their associated source data; and retrieval units towards units only storing the data matching their metadata combination. A memory management operation can thus selectively remove (from volatile/persistent memory) only the data originating from a particular online source[5]; while all data adhering to a given metadata combination can still be retrieved.

Finally, we note that when moving a storage unit from volatile to persistent storage, multiple options exist to store the source data on the file system[6]. We found that the straightforward option, namely saving each storage unit to a single file, leads to an impractically large number of files (i.e., # distinct metadata combinations X # origin sources). Instead, grouping the persistent data per removal or retrieval unit reduces the number of data files, and has other advantages as well. Grouping per removal unit optimizes memory management (i.e., storage, removal), since only a single file is affected. By grouping per retrieval unit, data retrieval is optimized, since a single retrieval only requires accessing one data file. In our experimental section (see Section 7), we discuss on the effects of these data grouping methods on performance.

5.3 Implementation

In this section, we discuss implementation issues related to the cache components.

- *Representing in-memory source data:* when assembling the final dataset for querying, an AndroJena RDF graph needs to be created on which the query is executed. Loading this query graph with separate data strings from each retrieved cache unit incurs a performance overhead. This was especially the case for Source Cache, with its coarse-grained cached data retrieval. Therefore, in Source Cache, each in-memory cache unit keeps its data in an AndroJena RDF graph. By optimizing the AndroJena library to efficiently combine AndroJena graphs, the final data assembly became much more efficient.

On the other hand, since the number of cache units in Meta Cache is comparably much higher (due to the higher amount of distinct metadata combinations), we found that keeping separate AndroJena graphs per cache unit caused too much memory overhead[7]. Therefore, cache units in Meta Cache still keep their data as a string.

- *Storage management:* in order to manage volatile and persistent storage space, which involves persistently storing or removing cached data when storage limits are exceeded, cached data sizes need to be accurately measured. Since no effective way to estimate runtime memory usage is available in Android, this is currently done by estimating source data sizes, which does not include implementation-specific data structures (e.g., Java object overheads). In our experiments, we compare the accuracy of this

[5] Removal of storage units is hereby propagated to their respective retrieval units, to update their internal pointers.

[6] Data is stored directly on the file system instead of a database, which would introduce unnecessary overhead.

[7] E.g., due to the internal indices used by AndroJena (see related work).

estimation with the actual memory usage, measured by analyzing Java heap dumps[8].

5.3 Cache validity

Various invalidation strategies exist to detect invalid, out-of-date information in client-server systems and mobile scenarios (see related work, Section 8). However, such strategies cannot be applied in our setting, where cached data does not originate from dedicated servers but from online files, stored on multiple, general-purpose web servers.

To accommodate our setting, we re-use web servers' existing functionality by relying on the cache support provided by the HTTP protocol (e.g., also used by proxy caches). For each retrieved source, the last download time and expiration time (indicated by the "Expires" header field), if available, is kept. If no expiration time is given, a configurable maximum life span is assigned to the source; which depends on the volatility of the dataset, and data freshness requirements of the application. Based on these two criteria, the system may also be configured to let the max. life span take precedence over the source expiry time, meaning the same life span will be assigned to all sources (e.g., in case data freshness is less important, or online sources are expected to evolve very frequently regardless of expiry times). To support timely validity checking, an ordered list of life spans / expiry times is kept by the system. In case the currently smallest time span has been exceeded, a background process checks the source's validity. For optimisation purposes, in case a number of sequential time spans are sufficiently similar, they are grouped to invoke the background process only once (cfr. Android AlarmManager API[9]). Checking source validity involves sending a conditional GET request to the source's web server, with its last download time filled into the "Last-Modified-Since" header field. If no change occurred, a 304 Not Modified header is returned, yielding only minimal data transfer overhead. Else, the updated source data is returned and used to update the cache.

6. The Semantic Web as an Distributed System, supporting the Open World Assumption

The vision of the Semantic Web is that of an open, interlinked web of machine-readable data, where semantic sources may publish information on anything identifiable by a resource URI. To that end, Semantic Web technology implements the Open World Assumption (OWA) which, contrary to the Closed World Assumption, states that no assumptions can be made on non-explicitly stated knowledge. As such, no data source may be assumed to be comprehensive and self-contained, and due to the distributed nature of the Semantic Web, additional information on resources, missing from the particular source, may be found in any other online source. Data sources are thus transformed from closed data silos to collaborating parties – each contributing their own data to the online Semantic Web knowledge base.

By supporting the distributed nature of the Semantic Web and the OWA assumption, our query service can provide fully integrated access to the Semantic Web. Supplying this support has two important consequences, which we discuss next.

6.1 Distributed type constraints

As mentioned, the OWA implies semantic sources are not self-contained, which also means that their comprised RDF resources may be described by other online sources. Regarding the query service, this means that new sources may specify different types for already processed data; possibly leading to previously indexed source metadata to become out-of-date. Consequently, resource types should ideally be tracked across online sources, whereby appropriate action is taken when incomplete source metadata is found. In doing so, we guarantee that all relevant query results are returned, for any online data composition. We call this process *type mediation*.

In particular, type mediation is applied in two cases; when new sources specify *additional* types for previously found resources, and when new sources specify *less* types than known for the comprised resources. In the former case, internal indices should be updated; and in the latter case, the extracted source metadata should be extended with the missing types. To keep track of resource types across sources, we rely on a resource index (implemented as a hashtable) linking found resources to their known types.

We note that online sources cannot be directly updated with the missing types, as data in our setting is captured in online files not under our control. As such, type mediation needs to be applied on new sources during the Source Indexing phase; as well as during the Data Query phase, on re-downloaded sources (due to cache misses) on which type mediation had already been applied. We finally note that, due to the different internal structures in the SIM and cache components, the type mediation process and resource index differ

[8] Due to their overhead, it is not possible to use heap dump analysis tools at runtime.

[9] https://developer.android.com/training/scheduling/alarms.html

for these components and their variants. For Meta Cache, we note that type mediation often involves loading previously cached data into memory, to update their associated metadata. As such, type mediation process will have a large impact on removal times, as the loaded cache units need to be moved back to persistent storage afterwards. In the experimental section (see Section 7), we study the effects of each type mediation process and index on performance, memory and data access.

Type mediation is a resource-intensive process, given the resource index and need for updating internal indices. Consequently, it contradicts req. 1, *Minimizing resource usage*. At the same time, we point out this resource index still consumes less memory compared to e.g., RDF stores, which often utilize multiple indexes to support fast querying (e.g., 3 for Androjena; and 6 for YARS [28] and HexaStore [27]). We also note that in some cases, type mediation may be safely disabled. By analyzing the online dataset, the existence of inconsistent typing can be ruled out. Alternatively, when there is control over the online sources (e.g., in closed-world systems), the source data can be automatically supplemented with missing resource types, ensuring consistent typing. Some applications also prefer fast, partial results over guaranteed completeness, especially in a Web setting (e.g., [31, 32]). Finally, in the real-world dataset used in our experiments, we observed that only a limited number of typing issues occurred (see Section 7).

It can be noted that related approaches integrating Semantic Web data suffer this problem, yet to the best of our knowledge, they do not consider it. For instance, the SemWIQ [20] and DARQ [19] query distribution systems do not update indexed resource types based on types found in other sources. As a result, related state of the art corresponds to the case where type mediation is disabled in our query service.

6.2 Inferring new types

A second important consequence of the Semantic Web OWA is that it allows new statements to be inferred, based on logical axioms specified in RDF schema definitions or OWL ontologies. For example, an ontology may contain property domain/range restrictions, which constrain the types of related subject/object resources. In case these type constraints are not explicitly stated in the RDF data, they may be inferred. This process is called *type inferencing*, and is supported by most RDF stores. Typically, these stores allow enabling/disabling inferencing to suit application needs and improve performance.

Analogously, our query service supports type inferencing and allows to enable/disable it. When enabled, type inferencing is applied during the Source Indexing phase to enrich extracted source metadata; and during the Data Query phase, to enhance the extracted search constraints of posed queries.

For this purpose, the Source Indexing phase is extended with the Ontology Manager (see Figure 1). This component provides inferencing support based on axioms from online schema definitions and ontologies. The Source Analyzer, responsible for extracting source metadata, employs the Ontology Manager to retrieve each found predicate's domain/range types, optionally including their subtypes. If encountered, these inferred types are added to the extracted source metadata[10], allowing more query-relevant data to be identified. Consider the following RDF snippet in Code 2 (namespaces omitted for brevity):

vub:thinker_in_all_states
  rdfs:label *"Thinker in all states"*.

vub:thinker_in_all_states
  geo:xyCoordinates
  "50.82242202758789,4.393936634063721".

Code 2. Example RDF snippet to illustrate type inferencing during Source Indexing.

The Source Analyzer contacts the Ontology Manager to obtain the domain type restriction of the **geo:xyCoordinates** predicate (specified in the GeoFeatures [33] ontology), namely geo:SpatialEntity; and subsequently extends the source metadata with this inferred type. Whenever a query is posed requesting all labels of geo:SpatialEntity resources, the vub:thinker_in_all_states resource will now be returned as a result; which would not have been the case without type inferencing.

During the Data Query phase, the Query Analyzer component, responsible for extracting query search constraints, leverages the same ontological knowledge. By utilizing the Ontology Manager, the Query Analyzer obtains each concrete predicate's domain and range types (possibly accompanied by their subtypes), and uses them to enhance the search constraints. In doing so, more irrelevant source data can be ruled out. Code 3 shows a query containing two triple patterns (namespaces omitted for brevity):

?restaurant **lgd:cuisine** *?cuisine* .
?restaurant **rdfs:label** *?label* .

[10] If inferred types are already materialized in the online dataset, type inferencing in this phase can be skipped.

Code 3. Example query to illustrate type inferencing during the Data Query phase.

Using the Ontology Manager, the Query Analyzer retrieves the domain type restriction of the **lgd:cuisine** predicate (as specified in the LGD ontology [34]), namely lgd:Restaurant, and adds it as a subject type to the two extracted search constraints. Since no type constraints were explicitly given, and **rdfs:label** is a much-occurring predicate, adding the extra inferenced lgd:Restaurant type has to potential to drastically improve selectivity.

To implement the two lightweight inferencing tasks mentioned above, we apply two mechanisms:

- To support retrieving all super types of a given type, we keep a hierarchy of Java objects, combined with a (hash) map linking type URIs to objects in the hierarchy;
- To retrieve all domain/range types of a given predicate, we keep an additional (hash) multimap linking predicates to their domain/range types.

Per source analysis, we additionally keep a cache of inferred domains/ranges and supertypes, as these are typically re-used inside a source. We found that these two straightforward mechanisms, combined with a temporary cache, greatly optimizate performance; compared to issuing queries on-the-fly on the ontology RDF graph to obtain the same information.

We note that, as for type mediation, type inferencing needs to be re-applied on re-downloaded sources during the Data Query phase; as it was not possible to update the online sources with the inferred types. Further, due to the increased size of the cache units, we note that removal times will be influenced as well.

7. Experimental evaluation

This section presents an elaborate experimental evaluation of the query service and its components. In these experiments, we apply a context-aware scenario, where the SCOUT mobile context-provisioning framework [11] plays the role of client. We extracted real-world semantic data sources from existing datasets (e.g., LinkedGeoData, DBPedia) to serve as an online experiment dataset.

These experiments focus on the difference aspects of the query service, and investigate:

- The utility of different amounts of source metadata in balancing fine-graininess of data retrieval with memory and processing requirements . This is studied for online source identification (SIM; Section 7.2) and local caching (Section 7.3).
- The impact of the novel Least-Popular-Sources strategy, with different configurations, on cache composition and query performance (section 7.4);
- The Open World Assumption features, namely type inferencing and mediation, and their positive effects on data access vs. memory and performance penalties (section 7.5).

All resources related to the experiments, including dataset and queries, can be found on [35] (queries are also included in Appendix A). Before going into detail on the experiments, we first describe the experiment setup and methodology below.

7.1 Experiment setup

This section outlines the setup for our experiments.

7.1.1 Device

The experiments were performed on an LG Nexus 5 (model LG-D820), with 2.26 GHz Quad-Core Processor, 2Gb RAM and 32Gb storage. We note that this device also runs the latest Android OS version (Android 5.1.1, Lollipop). Android apps obtain a maximum Java heap space of 192Mb.

7.1.2 Dataset

The semantic dataset used in the experiments consists of 5000 data sources, and has a total size of 526Mb; with an average size of ca. 107Kb, median size of 13Kb and standard deviation of 322Kb. The data sources were assumed not to change during the experiments, and were distributed across four different remote web servers.

The sources were extracted from 8 online datasets, some referenced on the Billion Triples Challenge (BTC) 2012 Dataset webpage [36]. The extracted data contain information on people (Timbl), places and things (Freebase, DBPedia, DataHub), shopping items (BestBuy RDF extract), geographical entities (LinkedGeoData, Geonames) and online news (NYTimes). An individual source groups data on a specific RDF resource; possibly obtained from multiple remote datasets and linked together using interlinks (released by the Linked Data initiative). Overall, the dataset references 191 ontologies.

Due to its re-use of existing online data, our experiment dataset can be considered as representative of real-world use cases. At the same time however, we note that different dataset compositions will influence certain results, such as for LPS (see Section 7.4.2) and type mediation (see Section 7.5.2.2). A systematic study of the query service using multiple, distinct dataset compositions is considered future work.

7.1.3 Query scenario

Our experimental evaluation is applied in a context-aware scenario, using the SCOUT context-aware application framework [11] as a client. As the user is moving around, SCOUT continuously discovers new physical entities in the user's vicinity (e.g., using a built-in mobile RFID reader), and extracts references to online semantic sources describing the particular entity (e.g., by reading URLs from RFID tags). To allow integrated querying over this gradually discovered semantic dataset, SCOUT dynamically passes detected source references to the query service.

For the experiments, five context-aware application queries were selected that request context-relevant data, covering the different types of data in our experiment dataset (e.g., geographical entities, people). Two queries return geographical data, for instance allowing to plot physical entities (e.g., shopping centers, airports) on a map. The other three queries return "interesting" physical entities in the vicinity (e.g., products for sale in an affordable price range), together with details and indication of relevance (e.g., manufacturer and user comments).

7.1.4 Methodology

All experiments were run on the aforementioned device (Section 7.1.1), using the extracted dataset (Section 7.1.2) and five selected queries (Section 7.1.3), in the following way:

Experiment initiation: before each experiment, the Android device was re-started to clear memory.

Query service phases: each experiment involved running the Source Indexing phase on all 5000 dataset sources, and the Data Query phase on the 5 experiment queries (unless stated otherwise).

Experiment runs: each experiment was run five times and the average processing times and battery usage were taken, to minimize the effect of external factors (e.g., OS background processes).

Below, we list the applied configurations, and make general notes on the measuring methods.

Cache configuration: the cache components were configured to use up to 75% of the dataset size for persistent storage[11], and 8Mb for in-memory storage[12]. As we deal with non-evolving sources, cache validation was disabled.

OWA features: where type inferencing was used, both domain/range constraints and subtype relations were leveraged to infer new types (only direct supertypes).

Memory usages: To accurately measure memory usage, snapshots of the Android Java heap were taken at runtime using Eclipse MAT [37], collecting the retained heap size of the revelant classes.

Energy consumption: we utilize the Android BatteryManager API to obtain the accurate energy consumption (in *Joules*) of query service processes. This involves sending an *Intent* each time energy usage needs to be calculated. After draining the battery purposefully, we found that mobile query processes had consumed 18226 Joules; any capacity percentage shown is relative to that number.

Dealing with network fluctations: to avoid network fluctuations influencing results, the query service retrieves RDF sources from persistent storage, whereby retrieval times were substituted by average download times from the sources' online locations (obtained by downloading 1000 random sources over 5 runs). In the same vein, ontologies referenced by the Ontology Manager were stored locally, and download times substituted in the same way. Evaluating the impact of network quality is subject of future work.

7.2 Experiment 1: Source Index Model

This experiment evaluates the impact of source metadata on selectivity when identifying query-relevant online sources. To that end, we compare different SIM variants, each keeping varying amounts of metadata: SIM1 only indexes predicates; SIM2 indexes predicates and subject types; and SIM3 indexes predicates, subject and object types. In addition, we consider the case where queries are executed on the entire dataset (i.e., native query engine performance). Since this experiment focuses on SIM selectivity, it does not include a local cache.

7.2.1 Experiment 1: Results

Source Indexing phase

Table 1 shows SIM memory usage for the total dataset; processing times, including metadata extraction, index update and download; and energy usage (between brackets).

	mem. size	*avg. dl.*	*processing*	
			extract & add	*total*
SIM1	1143	246 (*0.13J*)	40 *(0.04J)*	286 (*0.17J*)
SIM2	5789		54 *(0.06J)*	300 *(0.19J)*
SIM3	8893		57 (*0.07J*)	303 *(0.20J)*

Table 1. *SIM – Source Indexing*: sizes (Kb), processing times per source (ms) and energy usage (J)

[11] To force the necessity of cache removals.
[12] This relatively low limit was chosen since other components also take up memory (e.g., SIM, RDF graphs).

To process all 5000 sources, SIM1 consumed 4.7% battery capacity, SIM2 consumed 5.2%, and SIM3 consumed 5.3%. This includes downloading the 5000 sources, which consumes ca. 3,5% battery capacity.

Data Query phase

Table 2 illustrates source selectivity, by showing the number of identified (and potentially relevant) sources per query. In addition, it shows the total query resolution times and energy consumption.

	SIM1		**SIM2**		**SIM3**	
	#	*exec.*	*#*	*exec.*	*#*	*exec.*
Q1	2116	895310 (*272J*)	254	108199 (*33J*)	254	108270 (*33J*)
Q2	313	132289 (*40J*)	305	128906 (*39J*)	272	114949 (*35J*)
Q3	1293	546501 (*166J*)	319	134967 (*41J*)	319	134978 (*41J*)
Q4	1984	837748 (*254J*)	87	36803 (*11J*)	87	36805 (*11J*)
Q5	2146	932846 (*291J*)	256	132700 (*47J*)	256	132635 (*48J*)

Table 2. *SIM – Data Query*: selectivity (# sources), query times (ms) and energy usage (J)

To execute all 5 queries, SIM1 consumed 5.6% battery capacity, SIM2 and SIM3 consumed 0.9%.

Tables 3 to 5 show a breakup of the total query resolution times, including source identification[13] (*id*), data collection (*collect*) and query execution[14] (*execute*) times. For collection (collect), we separately indicate source download times (*dl*) and time to assemble the data into a query graph (*assemble*). We indicate the energy usage for source downloads and total resolution times.

SIM1					
		collect			
	id	*dl*	*assemble*	*execute*	*total*
Q1	730	520536 (*271J*)	372416	1628	895310 (*272J*)
Q2	16	76998 (*40J*)	55088	187	132289 (*40J*)
Q3	244	318078 (*166J*)	227568	611	546501 (*166J*)
Q4	219	488064 (*254J*)	349184	281	837748 (*254J*)
Q5	366	527916 (*275J*)	377696	26868	932846 (*291J*)

Table 3. *SIM1 – Data query*: times (ms) & energy usage (J)

SIM2					
		collect			
	id	*dl*	*assemble*	*execute*	*total*
Q1	161	62484 *(33J)*	44704	850	108199 (*33J*)
Q2	12	75030 (*39J*)	53680	184	128906 (*39J*)
Q3	20	78474 (*41J*)	56144	329	134967 (*41J*)
Q4	10	21402 (*11J*)	15312	79	36803 (*11J*)
Q5	46	62976 (*33J*)	45056	24622	132700 (*47J*)

Table 4. *SIM2 – Data query*: times (ms) & energy usage (J)

[13] This includes query analysis time as well.

[14] Query execution denotes the execution of the query on the already collected and assembled dataset.

SIM3					
		collect			
	id	*dl*	*assemble*	*execute*	*total*
Q1	209	62484 (*33J*)	44704	873	108270 (*33J*)
Q2	11	66912 (*35J*)	47872	154	114949 (*35J*)
Q3	27	78474 (*41J*)	56144	333	134978 (*41J*)
Q4	11	21402 (*11J*)	15312	80	36805 (*11J*)
Q5	75	62976 (*33J*)	45056	24528	132635 (*48J*)

Table 5. *SIM3 – Data query*: times (ms) & energy usage (J)

The case without SIM (i.e., native query engine performance) fails with an out-of-memory exception for all queries, and is thus not shown here.

7.2.2 Experiment 1: Discussion

Table 1 shows that memory overhead, data processing times and energy usage are larger for variants utilizing increasing amounts of metadata, which is to be expected. At the same time, the size of the largest SIM (SIM3) still only corresponds to a fraction of the dataset (ca. 1,7% of the 5000 sources), while the data processing overhead is almost negligible (< 60ms) for any SIM. In total, the largest SIM consumes ca. 6.2% battery capacity, mostly due to the required source downloads (4.3% download vs. 1.9% for processing). As such, our requirement of *Minimizing resource usage* (req. 1, Section 2) is met.

Table 2 indicates that SIM2 and SIM3 significantly improve source selectivity (ruling out 95% of the sources on average), thus adhering better to req. 3, *Minimizing online data downloads.* In contrast, out-of-memory errors occur when resolving any query without a SIM. Comparing SIM2 and SIM3, we only observe differences in selectivity for the 2nd query. Since this query restricts the object types of each triple pattern, SIM3 can utilize its additional metadata to increase selectivity. We also observe that the number of sources to download greatly influences energy usage. Capacity-wise, by ruling out many more sources, SIM2 and SIM3 reduce battery usage by 84% compared to SIM1, to a mere ca. 0,9% battery usage.

However, Tables 4 and 5 show SIM2 and SIM3 still incur an exceedingly high query resolution overhead (ca. 0.5 – 2.25 minutes). Most of this overhead occurs during data collection, which involves downloading the sources and assembling all source data into an (AndroJena) query graph (which requires parsing the data). As shown in the table, over half this time is spent on downloading the data (ca. 56%). Therefore, employing a local cache has the potential to greatly reduce collection overhead.

In conclusion, the SIM variants utilizing most source metadata (SIM2 and SIM3) represent the best solutions. These significantly increase selectivity, and thus improve query execution times and energy efficiency; while keeping data processing, memory usage and battery consumption during source processing in check. As such, regarding source identification, this confirms that source metadata indeed enables a balance between fine-grained data retrieval, and memory/ processing requirements. On the other hand, query resolution is clearly impractical, necessitating a local cache to reduce download times.

7.3 Experiment 2: Cache

This section evaluates the impact of caching on query service performance. We present an experiment comparing two different cache organizations: Source Cache, organizing cached data via origin source; and Meta Cache, grouping data via shared metadata. For Source Cache, the best performing SIM variant (SIM3) is employed for online source identification.

As Meta Cache performs both online source identification and local caching (see Section 5.1.2), it is deployed autonomously. For these experiments, Least Recently Used (LRU) is employed as removal strategy; Section 7.4 shows the effect of applying different removal strategies.

7.3.1 Experiment 2: Results

Source Indexing phase

In Table 6, we show the volatile memory and persistent storage space utilized by Source Cache and Meta Cache when serving the *full dataset*. We separately indicate the memory overhead of supporting data structures for the SIM and the cache (i.e., Java objects and internal indices), and the actual *payload* (i.e., the stored source data). For the latter, we further differentiate between the *measured* payload size (obtained via snapshots of the Java heap) and the *estimated* payload size (between brackets), which is approximated at runtime to dynamically manage memory space (see Section 5.3). For ease of reference, Table 6 also includes the corresponding SIM sizes, as Source Cache is used in combination with the SIM.

cache	*in-memory*			*persistent*
	SIM	*cache*	*payload*	
Source	8893	7198	6323 (8192)	405285
Meta	n/a	35821	16882 (7778)	426834

Table 6. *Cache – Source Indexing*: sizes (Kb).

In Table 7, we show the average data processing overhead and energy usage, resulting from inserting data into the cache (*insert*) and performing the LRU removal strategy whenever the cache is full (*removal*). Source Cache incurs an extra overhead of updating the SIM (*SIM*). Meta Cache incurs an extra overhead for extracting the different metadata combinations from the RDF sources (*extract*).

cache	*avg. dl.*	*add*			*removal*	*total*
		SIM	*extract*	*insert*		
Source	246 (*0.13J*)	65 (*0.07J*)	n/a	87 (*0.08J*)	1126 (*0.8J*)	1524 (*1.1J*)
Meta		n/a	81 (*0.09J*)	81 (*0.2J*)	170 (*0.3J*)	578 (*0.73J*)

Table 7. *Cache – Source Indexing:* processing times (ms) & energy usage (J) / source

To process 5000 sources, Source Cache consumes ca. 30% battery, while Meta Cache consumes ca. 20%. As mentioned before, downloading 5000 sources consumes ca. 3.5% battery capacity (included in the percentages shown above).

Data Query phase

In Tables 8 and 9, we show the total query resolution times and energy usages with their constituent parts. In particular, we distinguish between the following parts: 1) *query analysis*, which involves extracting search constraints; 2) *SIM access*, required by Source Cache for source identification; 3) *cache access*, which comprises retrieving cached data (*retrieval*) and downloading missing sources (*miss*); 4) *data assembly*, which involves assembling the retrieved data into a final query graph; and 5) *query execution*, where the query is executed on the collected query graph. For *retrieval*, we show the number of retrieved in-memory/persistent cache units, as well as the total retrieval time and energy usage. For *misses*, we indicate the total number of misses and amount of resulting source re-downloads (between brackets); accompanied by the total incurred download times and energy usage. The *data collect* part further shows the total number of returned triples (*#t*), thus illustrating the data retrieval fine-graininess. For Meta Cache, the number of generated type triples, required to make the type of cached resources explicit in the query dataset, is shown between brackets (see Section 5.1.2). To execute all 5 queries, Source Cache consumed 0.4% battery, while Meta Cache consumed 0.2%.

Source Cache											
	query analysis	*SIM access*	*cache access*					*data assembly*		*query exec.*	*total*
			retrieval		*miss*		*total*				
			#	time	#	*time*		*#t*	*time*		
Q1	133	30	8/246	4758 (*4.8J*)	0	0	4762 (*4.8J*)	25364	293	370 (*0.22J*)	5588 (*5.2J*)
Q2	23	3	0/19	334 (*0.38J*)	253 (253)	62238 (*34J*)	62572 (*34.4J*)	31706	296	133 (*0.09J*)	63027 (*34.5J*)
Q3	5	26	0/319	4720 (*5.1J*)	0	0	4724 (*5.1J*)	24396	291	264 (*0.17J*)	5310 (*5.3J*)
Q4	8	3	0/87	2732 (*2.6J*)	0	0	2732 (*2.6J*)	13295	152	70 (*0.04J*)	2965 (*2.7J*)
Q5	12	65	4/252	3249 (*5.6J*)	0	0	3251 (*5.6J*)	13392	158	24443 (*14J*)	27929 (*19.8J*)

Table 8. *Source Cache – Data Query*: query resolution (ms) & energy usage (J)

Meta Cache										
	query analysis	*cache access*					*data assembly*		*query exec.*	*total*
		retrieval		*miss*		*total*				
		#	*time*	#	*time*		*#t*	*time*		
Q1	63	1/609	2372 (*2J*)	176 (50)	12300 (*6.4J*)	14764 (*9.1J*)	1592 (19597)	529	342 (*0.2J*)	15698 (*9.4J*)
Q2	7	1/5	65(*0.15J*)	1 (32)	7872 (*4.1J*)	7938 (*4.3J*)	867 (1759)	56	112 (*0.08J*)	8113 (*4.3J*)
Q3	7	66/126	586 (*1J*)	24 (32)	7872 (*4.1J*)	8476 (*5.2J*)	1804 (25745)	639	214 (*0.2J*)	9336 (*5.5J*)
Q4	8	0/6	61 (*0.16J*)	0	0	61 (*0.16J*)	545 (648)	82	57 (*0.06J*)	208 (*0.23J*)
Q5	12	8/1	11 (*0.36J*)	0	0	11 (*0.36J*)	2914 (4292)	403	23657 (*14J*)	24083 (*14.3J*)

Table 9. Meta Cache – Data Query: query resolution (ms) & energy usage (J)

Table 10 shows maintenance times and energy usage resulting from cache access. This comprises 1/ updating the cache with new source data, in case missing data was downloaded (*update*); and 2/ running the removal strategy (*removal*), in case storage limits were exceeded. Since maintenance occurs after query resolution, it is not included in the access times. We note that these results heavily depend on the utilized removal strategy (see next section).

	Source Cache		**Meta Cache**	
	update	*removal*	*update*	*removal*
Q1	0	75137 (*44J*)	3316 (*16J*)	13 (*1.7J*)
Q2	1499 (*2.4J*)	76237 (*46J*)	456 (*1.4J*)	1 (*0.26J*)
Q3	0	47249 (*27J*)	847 (*4.4J*)	258 (*0.93J*)
Q4	0	37742 (*22J*)	0	1 (*0.01J*)
Q5	0	54736 (*31J*)	0	1 (*0.01J*)

Table 10. *Cache – Data Query*: maintenance (ms) and energy consumption (J)

Collectively, this maintenance process costs Source Cache ca. 0.9% battery capacity, and for Meta Cache ca. 0,04%.

7.3.2 Experiment 2: Discussion

As shown in Table 6, the Meta Cache supporting structures (*cache* column) take up significantly more memory (ca. factor 5) compared to Source cache, even combined with the SIM. This is in line with expectations, as Meta Cache requires 3 indices instead of just 1 for Source Cache. Given the number of distinct metadata combinations (24068), these indices also comprise more entries, and many more storage unit objects need to be kept (see Section 5.1.2).

However, this overhead still only takes up a fraction of the total dataset; 6,7% for Meta Cache, and 3% for Source Cache. We note that the payload size estimated at runtime (between brackets) is slightly higher than the actual payload size[15], due to lack of effective runtime memory analysis (see Section 5.3).

On the other hand, Table 7 shows that the overall processing overhead and energy consumption is much lower for Meta Cache. In particular, cache removal is much less costly for Meta Cache, reducing total processing and energy overhead. Meta Cache keeps smaller and more fine-grained cache units, leading to smaller storage and removal times. Also, we note that Meta Cache incurs an extra extraction step, which involves extracting source triples with their metadata (*extract* column). Although we optimized this process with factor 10 (see Section 4.2), it still takes up half of the processing overhead. Although source processing takes up 30% and 20% for Source and Meta Cache respectively, we note that this is a one time cost, and typically ensues over a longer period of time (thus spreading mobile resource usage over time).

The utility of local caching is clearly indicated by Tables 8 and 9. They show a dramatic decrease in total resolution times, compared to when no cache is used (see Table 2, 5): an average reduction of ca. 80% in total resolution for Source Cache, and ca. 90% for Meta Cache. Energy consumption is low (respectively 0,4% and 0.2% for all queries combined), down from

[15] This difference is higher for Meta Cache, since more cache unit objects are kept.

0.9% for SIM3 due to much less downloads. This reduced battery usage is especially apparent when looking at the individual energy usages in Tables 5 and 8-9.

In general, we observe that Meta Cache outperforms Source cache, especially regarding cache retrieval. For Meta Cache, retrieved cache units only comprise source data associated with the requested query metadata; resulting in more fine-grained retrieval, and thus lower cache retrieval times (see *retrieval – time*) and associated energy usage. This fine-graininess is further illustrated by the number of collected triples (see collect data - #t column). As such, Meta Cache adheres to our requirement of *Minimizing the query dataset* (Section 2, req. 2). In Source Cache, data is instead retrieved per origin source, whereby other, query-irrelevant data is also retrieved. Since most cache units are stored persistently, the majority of the data needs to be read from storage, significantly increasing retrieval time and energy usage. For Meta Cache, we further note that cached type statements do not require (persistent) retrieval, but are instead generated based on associated metadata (see Section 5.1.2). As before, these observations confirm that a balance between fine-grained retrieval and memory and data processing requirements can indeed be achieved by focusing on source metadata.

Despite improvements in data retrieval, cache misses have the ability to cause major problems for Meta Cache (see miss column). Indeed, cache misses for Q1, Q2 and Q3 cause a large number of source re-downloads, which result in worse performance for Meta Cache for Q1 and Q3. Since cache units keep data sharing the same metadata, typically with multiple origin sources, a cache miss requires re-downloading all related sources. This is especially problematic regarding energy usage, as downloading a source takes more energy than retrieving it locally (e.g., see Source Cache; Q5 retrieval time vs. Q2 miss time). In the following section, we evaluate a removal strategy aiming to mitigate this problem.

Although more triples are involved, data assembly times are slightly lower for Source Cache (avg. 238 ms vs. 342 ms). Source Cache retrieves source data in the form of AndroJena graphs, which can be very efficiently combined (see Section 5.3). On the other hand, assembly times for Meta Cache clearly depend on the number of triples. Query execution times are very similar for both Source and Meta Cache, and rather depend on the query complexity than the query dataset size. Regarding cache maintenance, Table 10 shows that cache removals are very costly for Source Cache, due to more coarse-grained cache units.

7.4 Experiment 3: Removal strategies

In this experiment, we evaluate our novel Least-Popular-Sources (LPS) removal strategy, designed to tackle the Meta Cache cache-miss problem described in the previous section. In particular, we study its ensuing cache composition and effects on query performance, and compare these findings to when a regular strategy is applied (in this case, LRU). The impact of using different factor weights in the LPS removal value calculation is investigated, as well as the impact of different persistent data groupings. We note that the download time factor is not considered here, since we aim to avoid network fluctuations influencing experiment results (see Section 7.1.4).

7.4.1 Experiment 3: Results

Source Indexing phase

Table 11 shows the Meta Cache memory sizes[16] for the total dataset, when respectively applying LRU and LPS. For each strategy, we indicate the *total* memory size taken up by Meta Cache, size of the *payload* (i.e., cache unit objects and source data), and the size taken up by extra supporting structures (e.g., indices) used by the removal strategy (*removal).*

Meta Cache			
removal strategy	*in-memory sizes*		
	total	*payload*	*removal*
LRU	35821	16882	1197
LPS	45777	27905	2325

Table 11. *Removal strategies – Source Indexing*: Sizes (Kb)

In Table 12, we show the resulting cache composition, focusing on "missing" data; i.e., data that was removed to clear persistent storage space. This includes missing keys (*keys*), which stand for metadata combinations associated with removed cache units; and missing sources (*sources*), which represent sources to be re-downloaded when a particular missing key is referenced (cache miss). The *distribution* column relates missing keys with missing sources; in particular, indicating the range of potential source re-downloads for missing keys. For instance, a missing key in range 1-10 incurs 1-10 source re - downloads in case the key is referenced. For LPS, we show the results for different weightings of the popularity factors (see Section 5.2.1) in the *strategy* column. These weightings were obtained by either considering only one of the two factors, or the sum of both factors, whereby the impact of one factor is

[16] These were accurately measured using the Eclipse MAT.

potentially reduced (i.e., divided by a power of 10). Note that since factor *f1/100 + f2* yields the same results as when f1 is not considered, it is left out.

LRU	LPS	
	removal-unit	**retrieval-unit**
170 (*0.3J*)	117 (*0.3J*)	204 (*0.4J*)

Table 13. *Removal strategies – Source Indexing*: replacement times (ms) & energy usage (J) / source

Table 13 summarizes the maintenance overhead. It only shows the overhead (time and energy usage) of running the removal strategy, since the extraction and insertion operations (see Table 7) are not influenced by removal strategies. For LPS, we further show the results for two potential persistent data groupings. For *retrieval-unit* grouping, persistent data sharing the same metadata is stored in a single data structure (i.e., file); for *removal-unit* grouping, persistent data is grouped based on their origin source. To obtain these values, the *f1+f2/100* popularity factor weighting was employed; since this struck the best balance between the number of missing keys and sources to be re-downloaded. To process 5000 sources, LPS *removal-unit* consumed 18% battery capacity, whereas LPS *retrieval-unit* consumed 20%.

Data Query phase

Table 14 presents times and energy usage resulting from cache access for LPS, as only these are influenced by the removal strategy. We refer to Table 9 for results related to LRU. As before, we differentiate between cache retrieval (*retrieval*) and misses (*miss*). Also, we show the total resolution time (also including constituent times not shown here). For LPS, each retrieval unit requires loading one or more storage units (see Section 5.2.2). Table 14 shows the number of retrieval units, with the amount of loaded storage units between brackets. In addition, the table shows retrieval times for each persistent data grouping method (a=removal-unit, b=retrieval-unit). As before, the cache miss part shows the total number of misses, accompanied by the resulting amount of sources to re-download (between brackets). We again assume the *f1 + f2/100* popularity factor weighting[29].

LPS – Meta cache						
	retrieval		*miss*		*cache*	
	#	*time*	*#*	*time*	*access*	*total*
Q1	784 (935)	a: 1301 b: 3594	5 (5)	1230	a: 2531 (*2.3J*) b: 4824 (*3.6J*)	a: 3547 (*2.6J*) b: 5854 (*3.8J*)
Q2	6 (272)	a: 383 b: 90	0	0	a: 383 (*1.3J*) b: 90 (*0.13J*)	a: 583 (*1.4J*) b: 289 (*0.25J*)
Q3	216 (957)	a: 1114 b: 695	0	0	a: 1114 (*2.1J*) b: 695 (*1J*)	a: 2081 (*2.3J*) b: 1536 (*1.2J*)
Q4	6 (467)	a: 587 b: 68	0	0	a: 587 (*0.6J*) b: 68 (*0.2J*)	a: 750 (*0.7J*) b: 214 (*0.3J*)
Q5	9 (2146)	a: 2226 b: 318	0	0	a: 2226 (*2.4J*) b: 318 (*0.9J*)	a: 28269 (*17J*) b: 25818 (*16J*)

Table 14. *LPS – Data Query*: query resolution (ms) and energy consumption (J)

To execute 5 queries, LPS *removal-unit* and *retrieval-unit* consumed ca. 0.1% battery capacity.

	LRU	LPS	
		replacement	
	replacement	*a*	*b*
Q1	13 (*1.7J*)	3220 (*2.1J*)	3466 (*3J*)
Q2	1 (*0.26J*)	10524 (*11J*)	20304 (*10J*)
Q3	258 (*0.93J*)	6011 (*7J*)	20980 (*13J*)
Q4	1 (*0.01J*)	6213 (*3.8J*)	11949 (*8J*)
Q5	1 (*0.01J*)	55202 (*90J*)	58655 (*40J*)

Table 15. *LPS – Data Query*: cache maintenance (ms) and energy consumption (J)

Finally, Table 15 shows the removal strategy times and energy usage from cache access. Both adding new data due to cache misses, as well as loading cached data into memory, may cause the memory limit to be exceeded, necessitating cache maintenance. For LPS, we again indicate these times for removal-level (*a*) and retrieval-level (*b*) grouping. Since maintenance occurs after query resolution, it is not included in the previously shown cache access times. We note that the maintenance process takes up ca. 0.6% battery for LPS *removal-unit*, and 0.4% for *retrieval-unit*.

7.4.2 Experiment 3: Discussion

Firstly, we observe that LPS incurs a larger memory overhead (see Table 11). By decoupling removal, storage and retrieval units, more cache unit objects need to be kept in-memory, and extra indices are needed to link these units. At the same time, while

strategy	*#keys*	*#sources*	*distribution*				
LRU	11058	1084	**1-10:** 10854	**10-50:** 196	**50-100:** 1	**100-250:** 6	**250-500:** 1
LPS *f1+f2*	9322	819	**1-10:** 9232	**10-50:** 82	**50-100:** 8	**100-250:** 0	**250-500:** 0
LPS *f1*	741	1586	**1-10:** 643	**10-50:** 60	**50-100:** 4	**100-250:** 21	**250-500:** 7
LPS *f1+f2/100*	961	1049	**1-10:** 857	**10-50:** 86	**50-100:** 4	**100-250:** 11	**250-500:** 3
LPS *f1+ f2/10*	2061	752	**1-10:** 1967	**10-50:** 85	**50-100:** 3	**100-250:** 6	**250-500:** 0
LPS *f2*	10314	785	**1-10:** 10224	**10-50:** 82	**50-100:** 8	**100-250:** 0	**250-500:** 0
LPS *f1/10+ f2*	10266	792	**1-10:** 10176	**10-50:** 82	**50-100:** 8	**100-250:** 0	**250-500:** 0

Table 12. *Removal strategies – Source Indexing*: removed data

presenting a 27% increase compared to LRU, this overhead still only makes up 9% of the total dataset.

Table 12 illustrates how LPS copes with the cache-miss issue of Meta Cache by affecting cache composition; in particular, by enabling a balance between the 1/ likelihood of cache misses, and 2/ the number of source re-downloads. The results show that as more preference is given to source-data popularity (f1), the total number of missing keys is minimized; decreasing the likelihood of cache misses. However, the number of missing sources increases as well, together with the amount of missing keys resulting in many source re-downloads (see ranges 100-250 and 250-500). When source-metadata popularity (f2) is preferred, the total number of missing sources decreases, and the number of source re-downloads per missing key is capped (i.e., no more outliers in ranges 100-250 and 250-500). However, the number of missing keys increases drastically, raising the chance of a cache miss. We note that the best weighting depends on the online dataset composition; including the number of distinct metadata combinations contained in sources (*f1*), and the extent to which metadata is shared across the online dataset (*f2*). As such, further research on this issue is needed. For our experiment dataset and queries, the weighting *f1+f2/100* yields the best balance. Table 13 further shows that, when grouping persistent data per removal-unit, memory management is slightly more efficient, since only one persistent file is affected per operation (see Section 5.2.2).

Reflecting the improved cache composition, significantly less cache misses are observed during query resolution (see Table 14). As a side-effect, this also increases the cache retrieval time, since more locally available (persistent) cache units are retrieved. Moreover, since retrieval and storage units are separated, a single retrieval likely results in accessing and combining data from multiple storage units, also increasing retrieval times. Nevertheless, overall query resolution times are reduced, in particular for those queries where cache misses presented a problem for Meta Cache (see Tables 9 and 14; Q1, Q2, Q3) compared to Source Cache. We also note that energy usage is much lower for these queries (0,1% for all queries); resulting from the fact that far less downloads are necessary. By applying LPS, Meta Cache now outperforms Source Cache for any query. Further, we observe that grouping the persistent data per retrieval-unit (option *b* in Table 14) optimizes retrieval time, since only one persistent file needs to be read per retrieval operation.

However, Table 15 again shows that cache maintenance occurring after query resolution, including performance times and energy usage, are much higher for LPS than LRU. Since the LPS removal unit is more coarse-grained (i.e., per origin source) compared to LRU (i.e., per metadata combination), larger removal penalties are incurred. Even when grouping persistent data per removal unit (column *a*), this maintenance results remain relatively high. For queries 1 – 4 this is avg. ca. 6,5s, with an outlier for query 5, which has a steeper overhead (ca. 59s). Further investigation and optimization of this process is future work (see Section 9). We nevertheless note that maintenance times and energy consumption for Meta Cache + LPS represent a significant improvement compared to the baseline approach, Source Cache + LRU (i.e., ca 57% improvement; see Tables 10, 15).

To conclude, the most optimal querying configuration is Meta Cache + LPS (*retrieval-unit*). For the relatively large experiment dataset, Meta Cache + LPS requires more memory (27%), processing time (20%) and slightly more energy (+0.1 J / source; although total battery usage is virtually equivalent) than LRU during the source indexing phase. These are one-time costs, and are typically incurred over a longer period of time (thus spreading mobile resource usage over time). Once set up, Meta Cache + LPS results in fast query execution times (6s; 0,3s; 1,5s; 0,2s, with an outlier of 25s for query 5), and low energy consumption (0,1% for all queries combined). These energy reductions are again most apparent when looking at individual energy usages in Tables 9, 14. For the outlier query, we note that the bulk of the resolution time (23s of 25s) is made up by the query execution time of the external RDF library, AndroJena. We thus conclude that for our experiment queries and dataset, this optimal configuration supplies realistic performance (barring the RDF library performance issues with query 5). However, LPS still incurs a high maintenance overhead after query execution, depending on the query (3s, 20s, 21s, 12s, 59s). Although these times already present a good improvement (ca. 57%) compared to the baseline approach (i.e., Source Cache), cache maintenance needs to be further optimized in future work.

7.5 Experiment 4: OWA features

This experiment evaluates the two Semantic Web Open World Assumption (OWA) features, type inferencing and type mediation. The best performing variant of the SIM and cache were considered, namely SIM3 and Meta Cache. We investigate the

improvements in data access and compare them to the incurred performance and memory overhead.

7.5.1 Experiment 4: Results

Source Indexing phase

Type inferencing can be applied at two places in the query service: on dataset sources (@source), and on posed queries (@query). Clearly, only type inferencing on sources influences performance during the Source Indexing phase.

Table 16 shows the overheads for SIM3 and Meta Cache when type inferencing is enabled. Firstly, it shows the increased *memory size* for the total dataset. Also, the extra computational overhead and energy usage of type inferencing is shown (*performance)*, including the inferencing time itself (*infer*) and ontology retrieval time (*retrieval*). We further show the removal strategy time, which is influenced by the increased cache unit size due to type inferencing (see Section 6.2). We note that, due to the exceedingly high amount of energy consumed by cache removal, the mobile battery was drained after 4524 sources.

Type mediation requires resource information to be tracked, such as types. To index this information, we keep a separate resource index (see Section 6.1). Table 17 shows the computational overhead and energy usage of type mediation for SIM3 and Meta Cache (*mediation*), together with the memory consumed by the resource index (*index size*). As the type mediation processes differ for these components, different index sizes and mediation results are incurred. We again show the removal strategy time, which is likewise influenced by type mediation (see Section 6.1). As was the case before, cache removal drained the mobile battery after 4464 sources.

	memory size (Kb)	*performance (ms)*		
		infer	*retrieval*	*removal*
SIM3	60549	11	60	n/a
Meta Cache	49364			3024 (*15446J – 3.4J / src*)

Table 16. *Type inferencing – Source Indexing*: memory (Kb), performance (ms) overhead and energy usage (J)

	index	*performance (ms)*	
	size (Kb)	*mediation*	*removal*
SIM3	71238	996	n/a
Meta Cache	63846	2264	3986 (*6636J – 1.5J / src*)

Table 17. *Type mediation – Source Indexing*: index size (Kb), performance (ms) overhead and energy usage (J).

Data Query phase

Regarding type inferencing, we consider three cases during querying; applying type inferencing on posed queries (*@query*), on dataset sources (*@source*), and on both *(@both*). Table 18 shows the effects on data access, indicating the number of query results (*res*) as well as the amount of sources identified by the SIM[17] (*src*). For ease of reference, the table also shows the original selectivity (*original*). In case the results differ from the original, the new results are shown in bold.

	original		*@query*		*@source*		*@both*	
	res	*src*	*res*	*src*	*res*	*src*	*res*	*src*
Q1	4	254	**0**	**49**	4	254	4	**215**
Q2	272	272	272	**271**	**658**	**313**	**658**	**313**
Q3	319	319	**0**	**0**	319	319	319	319
Q4	77	87	77	87	77	87	77	87
Q5	148	256	**0**	256	148	256	148	256

Table 18. *Type inferencing – Data Query*: data access.

Table 19 shows the type inferencing overhead for both components during querying. As type inferencing needs to be re-applied to re-downloaded sources (see Section 6.2), this process also incurs a query-time overhead @source. We note that, since the source indexing phase drained the battery, no battery usage data is available for the data query phase.

	@query	**SIM**	**Meta Cache**
		@source	*@source*
Q1	174	45051	341
Q2	100	484646	420
Q3	128	383611	173
Q4	84	3743	0
Q5	156	120774	0

Table 19. *Type inferencing – Data Query*: data retrieval (ms).

For type mediation, Table 20 illustrates the effects on data access, indicating the new SIM source selectivity (*src*) and new amount of query results (*res*) (values differing from the original in bold). Also, the table indicates the performance overhead during querying (*synchronization*). Comparable to type inferencing, the original contents of re-downloaded sources need to be synchronized with mediated resource types (see Section 6.1). As before, since the source indexing phase drained the battery, no battery usage data is available for the data query phase.

	selectivity		*synchronization*	
	res	*src*	***SIM3***	***Meta Cache***
Q1	4	254	1104	113
Q2	**273**	272	1413	58
Q3	319	319	1190	151
Q4	77	87	235	40
Q5	148	256	1008	17

Table 20. *Type mediation – Data Query*: data access (ms)

7.5.2 Experiment 4: Discussion

This section discusses the effects of applying the OWA features.

[17] Although the Meta Cache selectivity is also influenced, the increase in selectivity is most apparent for the SIM.

7.5.2.1 Type inferencing

From Table 16, we observe that type inferencing results in large memory usage. While the computational overhead of type inferencing itself is acceptable, it incurs a very high removal time (and associated high energy usage, draining the battery after processing 4524 sources); due to the increased size of the cache units. This contradicts our requirement of reduced resource usage. As such, we conclude that, in our mobile query service, type inferencing is unfeasible at this point.

Regarding query resolution, Table 18 shows that applying type inferencing on posed queries (@query) leads to the search constraints ruling out more sources (Q1, Q2, Q3), although results are no longer returned for Q1, Q3 and Q5. On closer inspection, extra query type constraints are inferred for those queries that are not found in the online dataset. Typically, content authors do not exhaustively type RDF resources; an issue that can be resolved by additionally applying type inferencing on sources. In that case (@both), the same inferred types are added to the source metadata, resolving the issue. Furthermore, many more query results (658) are now returned for Q2, thus enhancing data access. Finally, compared to only applying source type inferencing (@source), we observe that additionally enabling query type inferencing (@both) improves data selectivity for Q1.

The above indicates that type inferencing should be applied on both queries and sources (@both). Table 19 shows that for Meta Cache, type inferencing yields an acceptable overhead, but exceedingly high processing times for SIM. This results from re-applying type inferencing to all identified sources (SIM), which are more numerous than cache-missed sources. Since online data sources are not under our control, they cannot be updated with inferred types. Locally storing inferred types for online sources could mitigate the problem to some extent, and is considered future work. We also note that inferred types may already be materialized in the online dataset (see Section 6.2); if so, type inferencing @source is unnecessary.

Given our observations regarding source indexing, we conclude that type inferencing, when aiming to ensure completeness of query results, is currently not feasible in our query service. Analogously to RDF stores, type inferencing can be switched on/off to suit dataset composition, device capabilities and app requirements.

7.5.2.2 Type mediation

Table 17 shows that, just like with type inferencing, type mediation results in large memory usage. At the same time, computational overhead is problematic as well; regarding both type mediation overhead and removal overhead. Since type mediation involves continuously loading previously stored cache units into memory (see Section 6.2), it incurs an exceedingly high removal time and energy usage (draining the battery, as was the case for type inferencing, after 4464 sources).

Both these observations contradict req. 1, *Minimizing resource usage*, and makes type mediation currently impractical for mobile devices for our current query service. Analogous to type inferencing, type mediation also incurs a query-time overhead called synchronization (see Table 20). In this process, identified sources (SIM) or cache-missed sources (Meta Cache) are synchronized with the previously mediated types. As before, this process is necessitated by our setting where online sources cannot be updated. However, overheads resulting from this process can be considered acceptable for Meta Cache and SIM.

We further observe only a small impact on selectivity and data access, with the same SIM source selectivity and only one extra query result (Q2). In particular, this extra result concerned an RDF resource that was referenced but not typed in a first source, and then typed in a second source. For our real world dataset, situations where RDF resources were found in multiple sources occurred 2097746 times, and only in 1,7% did these sources specify different resource types, thus necessitating type mediation. Clearly, the dataset composition will impact the number of occurrences. At the same time, as our experiment dataset was extracted from real-world sources, this may be considered an indication for other datasets as well. An a priori analysis could determine whether type mediation is required, whereby the process could be disabled to reduce memory and processing overhead. This is subject of future work.

8. Related work

Currently, a number of mobile RDF stores exist to access and manipulate locally stored RDF data, including AndroJena [25], RDF On The Go [15], and i-MoCo [16]. Analogous to our query service, the MobiSem Context Framework [12] aims to supply transparent and integrated access to multiple online Semantic Web sources. The framework continuously and pro-actively replicates Semantic Web data from pre-configured online datasets, based on their relevance to the user's context, and supplies programmatic access to the local data. Such pro-active data selection avoids downloads at query-time, yet it is

necessarily domain-specific, and cannot support arbitrary application queries. In contrast, our query service is re-active and thus supports any scenario encapsulated by application queries; at the cost of potential download overhead at query time.

Query distribution approaches likewise supply integrated query access across multiple online datasets. As opposed to retrieving relevant data and querying it locally, these systems distribute query execution across dataset query endpoints. In particular, they divide queries into subqueries, each of which is executed on relevant datasets; and afterwards integrate the results. Such approaches relieve clients of resource-intensive query resolution, and are well suited to query large datasets outfitted with online query endpoints. However, they are not suitable for semantic data not residing behind a query endpoint, which is the focus of our query engine.

To identify query-relevant datasets, as well as optimize query distribution, query distribution systems typically rely on indices. The Distributed ARQ (DARQ) [19] and Semantic Web Integrator and Query Engine (SemWIQ) [20] systems each keep an index with summary info on each dataset, including found predicates, classes (SemWIQ) and resource patterns (DARQ), indicating which subjects and objects occur together with found predicates. Statistical information is kept as well, which is used to further optimize query distribution. The authors in [24] further index predicate paths found in datasets, allowing a more accurate identification of relevant datasets. In settings where datasets are under third-party control, keeping these indices up-to-date is paramount. The aforementioned query distribution approaches, as well as our query service, tackle this issue by focusing mainly on schema-level information (e.g., classes and predicates), as it can be assumed that schema-level changes will occur less often. The Adaptive Distributed Endpoint RDF Integration System (ADERIS) system [38] aims to avoid this issue by keeping only limited summary data, and instead collecting runtime selectivity estimates. Notwithstanding their similarity in using source metadata for indexing purposes, we note that none of these approaches explicitly takes the Open World Assumption into account, and thus do not guarantee query result completeness.

Many RDF stores focus on keeping extensive indices to speed up access to RDF data, trading index space and update efficiency for retrieval time. AndroJena is a port of the well-known Jena RDF store to the Android platform. To speed up query access, this store uses 3 hash tables, respectively indexing the subjects, predicates and objects of RDF triples. Depending on the concrete terms specified in the query, AndroJena selects between these hash tables. Our Meta Cache utilizes a similar index structure for quick data retrieval. However, since the Meta Cache indices keep schema-level information instead of instances, they contain significantly less entries. Similar to AndroJena, other RDF stores also trade memory space to optimize data access. The Yet Another RDF Store (YARS) system [28] keeps 6 indices to cover all potential triple access patterns. HexaStore [27] similarly relies on a sextuple indexing scheme to cover each potential triple access pattern. Aside from their higher memory usage, caused by having multiple instance-based indices, these approaches also have higher update and insertion costs, since all indices need to be updated [27]. Analogous to our query service, both systems apply dictionary encoding to reduce storage space and optimize query processing. We note that two of the mobile RDF stores mentioned at the beginning of the section, namely RDF On The Go [15] and i-MoCo [16], are respectively built on top of YARS and Hexastore.

Most caching approaches are based on client-server architectures, where data can be retrieved on-demand from the server and clients cache the data for later re-use [22]. In case of a cache miss, the missing data is directly obtained from the server. Query caching presents a particular type of client-server caching, whereby query results are cached and later re-used by other queries, by using query folding techniques [39]. To deal with cache misses, the system generates a remainder query to retrieve missing data from the server. These kinds of approaches cannot be directly applied in our setting, where data does not originate from a particular online server, but is instead spread across online files. Regarding cache replacement, ample work has been put in developing policies for mobile settings. Such policies typically rely on semantic locality, which is based on general properties and relations of data items. For instance, in [22], semantic locality indicates that query results, associated with physical locations closest to the user, will be frequently referenced. Similarly, the Furthest-Away-Replacement (FAR) policy [23] assumes that cached data, which is located in the user's movement direction and currently nearby, will be frequently referenced. As before, we opted for a replacement policy that is instead domain-independent, and focuses on dealing with our particular querying scenario where data is captured in online files.

Finally, various invalidation strategies exist to detect invalid, no longer up-to-date information in client-server architectures and mobile scenarios. For example, the Selective Adaptive Sorted (SAS) invalidation strategy [40] ensures that updates on data items on the server are reflected on the mobile device. In [41, 42], the authors present location-dependent cache invalidation, which ensures validity of location-specific cached data retrieved from information services. As before, such strategies are not suitable in our setting, where data does not originate from a single, special-purpose server. Instead, we rely on the built-in cache support of HTTP, which is typically also used by proxy caches.

9. Conclusions

We presented a general-purpose mobile query service, which supplies client applications with integrated querying capabilities across a currently untapped part of the Semantic Web; consisting of large amounts of small sources, namely RDF files and the growing set of annotated websites. Mobile clients are hereby able to outline and dynamically extend their relevant selection of online semantic data, according to the application scenario and requirements.

Our solution is conceived according to a number of challenges occurring in this particular mobile querying scenario, as well as their ensuing requirements. It involves 1/ fine-grained identification of query-relevant online sources, and 2/ locally caching data for later re-use. In order to reconcile fine-grained data selection, either during online source identification or cached data retrieval, with memory and processing usage, we developed source identification and caching components leveraging the semantics of RDF(S)/OWL data. To fully evaluate the effect of source metadata in realizing this goal, we developed and evaluated several variants for each component. Regarding source identification, three Source Index Models were implemented; each maintaining increased amounts of metadata. We further explored two cache variants, Source Cache and Meta Cache, which respectively organize cached data based on origin source and shared metadata. To optimize the query service for large amounts of small, online semantic sources, we introduced a removal strategy called Least-Popular-Sources (LPS). Our query service further explores supports for the Semantic Web's distributed nature and OWA by keeping indexed metadata up-to-date, in light of newly discovered sources (type mediation); and inferring new metadata to potentially identify additional query results (type inferencing).

An experimental validation, using a real-world dataset in a context-aware application scenario, confirmed the utility of source metadata to reach the aforementioned goal; namely, balancing high data selectivity with memory/performance overhead. We found that Meta Cache, combined with the LPS (retrieval-unit) removal strategy, supplied the best performance. After an initial source indexing phase, which incurs a one-time, noticeable cost in our experiments (but will usually be spread over time), we show realistic query performance and energy consumption. However, we also observed that this configuration incurs notable maintenance overhead after query execution; which is steep in some cases. Finally, type inferencing, and to a lesser extent type mediation, proved useful in improving data access by returning additional query results. However, the experiments showed they currently exhibit impractical performance and energy usage; mostly resulting from problematic cache maintenance times.

Future work includes investigating how cache maintenance for source-based replacement, which involves persistently storing large amounts of data, can be made more efficient. Optimizations for our OWA features, including storing previously inferred types (e.g., using incremental reasoning to cope with dataset updates [43]), and analyzing the online dataset to determine the necessity for type mediation, are also considered future work. We further aim to consider issues such as the composition of datasets and the impact of network delays in future experiments. Finally, additional efforts are needed to fully support semantic data exploration in the "wild". For instance, existing interlinks (i.e., *owl:sameAs* statements) can be leveraged to determine equivalence between two resources with different URIs; and existing ontology matching approaches can be applied to align heterogeneous ontologies.

10. References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. Int. J. Semant. Web Inf. Syst. 5, 1–22 (2009).
2. Sindice, http://sindice.com/.
3. Schema.org, http://schema.org/. Access date: 1/3/2015.
4. Adida, B.: hGRDDL: Bridging microformats and RDFa. Journal of Web Semantics 6, 54–60 (2008).
5. Web Data Commons, http://webdatacommons.org/structureddata/.
6. 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).
7. Zander, S., Chiu, C., Sageder, G.: A computational model for the integration of linked data in mobile augmented reality applications. Proceedings of the 8th International

Conference on Semantic Systems. pp. 133–140. ACM, New York, NY, USA (2012).
8. Ziegler, C.: Semantic web recommender systems. In Proceedings of the Joint ICDE/EDBT Ph.D. Workshop 2004 (Heraklion. pp. 78–89. Springer-Verlag (2004).
9. Wilson, M., Russell, A., Smith, D.A., Owens, A., Schraefel, M.C.: mSpace Mobile: A Mobile Application for the Semantic Web. User Semantic Web Workshop, ISWC2005 (2005).
10. Becker, C., Bizer, C.: DBpedia Mobile: A Location-Enabled Linked Data Browser. LDOW. CEUR-WS.org (2008).
11. 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).
12. 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).
13. Keller, C., Pöhland, R., Brunk, S., Schlegel, T.: An Adaptive Semantic Mobile Application for Individual Touristic Exploration. HCI (3). pp. 434–443 (2014).
14. 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).
15. 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).
16. Weiss, C., Bernstein, A., Boccuzzo, S.: i-MoCo: Mobile Conference Guide Storing and querying huge amounts of Semantic Web data on the iPhone-iPod Touch. Semantic Web Challenge 2008 (2008).
17. Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: Transparent Mobile Querying of Online RDF sources using Semantic Indexing and Caching. In: Proceedings of the 12th International Conference on Web Information System Engineering. pp. 185–198. Springer-Verlag, Sydney, Australia (2011).
18. Bolchini, C., Curino, C., Schreiber, F.A., Tanca, L.: Context Integration for Mobile Data Tailoring. Proceedings of the 7th International Conference on Mobile Data Management. p. 5. IEEE Computer Society, Washington, DC, USA (2006).
19. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. Proceedings of the 5th European semantic web conference on The semantic web. pp. 524–538. Springer-Verlag, Berlin, Heidelberg (2008).
20. 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 (2008)
21. Bolchini, C., Quintarelli, E.: Filtering mobile data by means of context: a methodology. Springer-Verlag, LNCS 4278. pp. 1986–1995 (2006).
22. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. Proceedings of the 22th International Conference on Very Large Data Bases. pp. 330–341. San Francisco, CA, USA (1996).
23. Ren, Q., Dunham, M.H.: Using semantic caching to manage location dependent data in mobile computing. Proceedings of the 6th annual international conference on Mobile computing and networking. pp. 210–221. ACM, New York, NY, USA (2000).
24. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.: Towards distributed processing of RDF path queries. Int. J. Web Eng. Technol. 2, 207–230 (2005).
25. AndroJena, https://code.google.com/p/androjena/.
26. Phuoc, D. Le, Parreira, J.X., Reynolds, V., Hauswirth, M.: RDF On the Go: RDF Storage and Query Processor for Mobile Devices. In: ISWC Posters&Demos. (2010).
27. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. 1, 1008–1019 (2008).
28. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. Presented at the (2005).
29. Paret, E., Van Woensel, W., Casteleyn, S., Signer, B., De Troyer, O.: Efficient Querying of Distributed RDF Sources in Mobile Settings based on a Source Index Model. In: Proceedings of the 8th International Conference on Mobile Web Information Systems. pp. 554–561. Elsevier, Niagara Falls, Canada (2011).
30. SPARQL Parser library, http://sparql.sourceforge.net/. Access date: 1/3/2015.
31. Shanmugasundaram, J., Tufte, K., DeWitt, D., Maier, D., Naughton, J.F.: Architecting a Network Query Engine for Producing Partial Results. World Wide Web Databases. 1997, 58–77 (2001).
32. Raman, V., Hellerstein, J.M.: Partial Results for Online Query Processing. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. pp. 275–286. ACM, New York, NY, USA (2002).
33. GeoFeatures, http://niche.cs.dal.ca/materials/ontologies/lgd-ontology.nt.bz2. Access date: 1/3/2015.
34. LGD Ontology, http://niche.cs.dal.ca/materials/ontologies/geoFeatures.owl. Access date: 1/3/2015.
35. Van Woensel, W.: Online Documentation, http://niche.cs.dal.ca/materials/mobile-query-service.
36. Billion Triples Challenge 2012, http://km.aifb.kit.edu/projects/btc-2012/. Access date: 1/3/2015.
37. Eclipse Memory Analyzer (MAT), http://www.eclipse.org/mat/. Access date: 1/3/2015
38. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: Adaptive Integration of Distributed Semantic Web Data. Databases Networked Inf. Syst. 5999, 174–193 (2010).
39. Ren, Q., Dunham, M.H., Kumar, V.: Semantic Caching and Query Processing. IEEE Trans. Knowl. Data Eng. 15, 192–210 (2003).
40. Safa, H., Artail, H., Nahhas, M.: A cache invalidation strategy for mobile networks. J. Netw. Comput. Appl. 33, 168–182 (2010).
41. Xu, J., Tang, X., Lee, D.L.: Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments. IEEE Trans. Knowl. Data Eng. 15, 474–488 (2003).
42. Zheng, B., Lee, W.-C., Lee, D.L.: On semantic caching and query scheduling for mobile nearest-neighbor search. Wirel. Netw. 10, 653–664 (2004).
43. Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, Josiane Xavier Aroyo, L., Noy, N., Welty, C., Janowicz, K.: DynamiTE: Parallel Materialization of Dynamic RDF Data. 12th International Semantic Web Conference. pp. 657–672. , Sydney, Australia (2013).