# Querying Heterogeneous Personal Information On The Go*

Danh Le-Phuoc, Anh Le-Tuan, Gregor Schiele, and Manfred Hauswirth

INSIGHT Centre for Data Analytics
National University of Ireland, Galway.

**Abstract.** Mobile devices are becoming a central data integration hub for personal information. Thus, an up-to-date, comprehensive and consolidated view of this information across heterogeneous personal information spaces is required. Linked Data offers various solutions for integrating personal information, but none of them comprehensively addresses the specific resource constraints of mobile devices. To address this issue, this paper presents a unified data integration framework for resource-constrained mobile devices. Our generic, extensible framework not only provides a unified view of personal data from different personal information data spaces but also can run on a user's mobile device without any external server. To save processing resources, we propose a data normalisation approach that can deal with ID-consolidation and ambiguity issues without complex generic reasoning. This data integration approach is based on a triple storage for Android devices with small memory footprint. We evaluate our framework with a set of experiments on different devices and show that it is able to support complex queries on large personal data sets of more than one million triples on typical mobile devices with very small memory footprint.

**Keywords:** mobile database, personal information system, RDF store

## 1 Introduction

The availability of an up-to-date, comprehensive and consolidated view of a user's social context not only enables novel applications such as distributed social networks [16], semantic life [8], or a semantic desktop [18] but is increasingly becoming an essential requirement for many mobile applications. As an example, consider a typical mobile user who has access to contact information of his acquaintances via Facebook, LinkedIn, Google+ and his phonebook. Each of these data sources may contain different types of information about this user, e.g., personal and professional information, phone numbers or message and call histories, and they may exhibit different levels of quality. Consequently, a contact management application should be able to link and integrate all this information

* This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289 and by Irish Research Council under Grant No. GOIPD/2013/104 and European Commission under Grant No. FP7-287305 (OpenIoT) and Grant No. FP7-287661 (GAMBAS) and Grant No. FP7-ICT-608662 (VITAL)

and automatically extract and integrate the right pieces of information from the whole set of data sources available. Similarly, a messaging widget should be able to integrate the messages from different services in order to track the user's conversations across all messaging service platforms.

However, the creation and continuous maintenance of such a view is a challenging task. The reason for this is twofold: First, despite the steady increase in computation and communication capabilities, mobile devices are battery powered. As a result, developers typically spend a considerable fraction of their development time on minimizing the amount of computation and bandwidth utilization to reduce the impact of their applications on the device's energy profile. Second, despite the popularity of some mainstream social networking services, the creation of a truly comprehensive view on the user's context usually requires the integration of considerable amounts of data from a user-specific set of services. As a result, developers must provide mechanisms to deal with the integration of complementary as well as overlapping and possibly inconsistent data sets.

Existing solutions typically use one of the following two approaches: Either they may use a powerful and well-connected cloud infrastructure to perform the data integration [12] or they may focus on the integration of an application-specific set of data types from a (possibly) limited set of services [13]. The first approach requires the provisioning of access credentials to the centralised/cloud-based data integration infrastructure which may then access, process and store the user's information. This remotely processing approach for mobile applications raises several privacy and security concern as security credentials leave the device and privacy is given up (entrusted to the cloud/remote servers without control by the users and without means to enforce it). Therefore, granting access on the mobile device is under the full control of the user is desired in ongoing security and privacy debates in many countries in respect to the cloud. To this end, the second approach is the preferable choice. However, it is not cost-effective as it requires developers to repeatedly make complicated design decisions. Furthermore, it may be also inefficient, especially in cases where multiple applications require access to the same data resulting in duplicate data retrieval and integration.

In this paper, we present an alternative approach for data integration by introducing a comprehensive framework that takes care of data retrieval, identity consolidation, disambiguation, storage and access, locally on mobile devices. To reduce privacy and security concerns, the framework does not require any remote storage and processing. It is solely executed on the mobile device of a user. In contrast to application-specific approaches, our framework is generic with respect to the supported types of data. It is extensible with respect to the supported services and it is open with respect to application support. To achieve this, the framework (1) leverages Linked Data to facilitate the storage of arbitrary types of data, (2) employs a plug-in model to connect to different services and (3) provides a generic query processor with support for SPARQL to be open with respect to application support. As a validation of the usefulness of the

framework and to verify the efficiency of the framework, we present the results of an extensive experimental evaluation.

The remainder of this paper is structured as follows: In the next section we describe our approach for data consolidation and integration on mobile devices. After that we present the design and implementation of our framework, and evaluate its performance. Finally, we discuss related work and finish the paper with our conclusions and discuss directions for possible future work.

## 2 Integration of heterogeneous personal information

To integrate the personal data from different data sources, several approaches proposed a unified data model for transforming heterogeneous data formats to RDF driven by agreed-upon vocabularies (FOAF, SIOC, vCard,etc) [2]. RDF statements are used to link and describe people, their social relationships, the content objects relevant to them, etc. However, a person can have multiple identifiers (IDs) on different data spaces. When they are integrated in a single data space, these IDs have to be interlinked and unified to represent a unique person. To uniquely identify someone across various data spaces, there are some rules that have to be set to infer and ensure uniqueness of that person. Along with some explicit properties like *owl:sameAs*, there are some implicit rules defined from properties indicating that two IDs are "talking" about the same person [2]. For instance, in practical, an "*inverse-identification*" property is used as an indirect identifier, e.g., foaf:phone, foaf:mbox_sha1sum. Therefore, having multiple identifiers poses several challenges for aggregating personal data from heterogeneous data spaces to store in RDF and to make it useful on resource constrained devices.

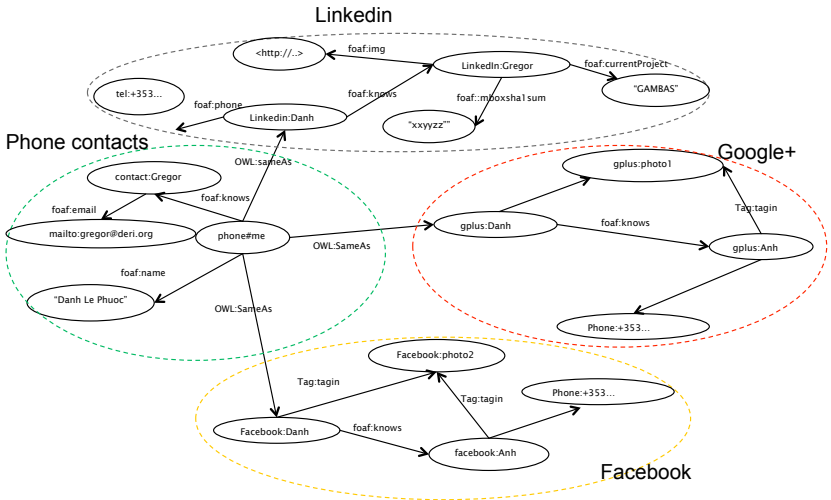



Fig. 1: Simple RDF graph integrated from data silos

To demonstrate why it is challenging to enable a unified, integrated view of heterogeneous personal information sources on mobile devices, let us take a closer look on the example depicted in Figure 1. The data shown in this figure

is acquired and transformed to RDF from Facebook, Google+, LinkedIn and phone contacts of a user's mobile phone in a similar fashion as proposed in [16, 2].

The explicit *owl:sameAs* statements are added to link a user's IDs from different data spaces. In addition, two RDF nodes, *facebook:Anh* (Facebook identifiers) and *gplus:Anh* (Google+ identifiers), represent one person because they have the same inverse-identification property *foaf:phone* with the same value $<$ *phone* : +35389... $>$. Similarly, two RDF nodes, *linkedin:Gregor* (LinkedIn identifier) and *phone:Gregor* (identifier given by the phone's contact application) also represent one person because the *sha1sum* value of his email has the same value as his *foaf:mbox_sha1sum* in LinkedIn. In essence, this RDF graph implicitly represents "different" pieces of information of three people who have different RDF statements attached with different RDF nodes representing each of them. However, if we store the simple RDF reification[1] of this graph in a standard RDF store, the SPARQL query processor will not be able to return complete information about a person. For instance, the query *"SELECT ?friends WHERE {phone:me foaf:knows ?friend}"* can only return one friend with the identifier *contact:Gregor* from the explicit statement in the phone contacts. Standard SPARQL is not able to infer the implicit statement {*phone:me foaf:knows facebook:Anh*} because *phone:me* and *facebook:me* is the same person.

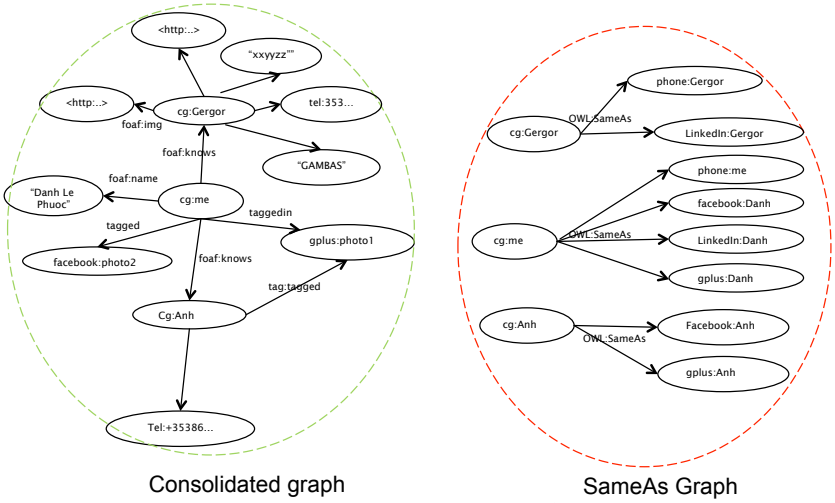



Fig. 2: Consolidated and SameAs graph

The solution for this problem is to use entailment regimes[2] instead of simple entailment in the above graph. This requires a modification in the SPARQL query processor to employ a reasoner to infer implicit RDF statements for basic graph pattern matching operators. However, this approach is not practical because it needs a considerable amount of memory and a fairly powerful CPU for

[1] `http://www.w3.org/TR/rdf-primer/`
[2] `http://www.w3.org/TR/sparql11-entailment/`

the reasoning process. Another alternative solution is to use an ID consolidation approach [12, 6] to compute all implicit RDF statements, then store them in an RDF store and query it with a standard SPARQL query processor. However, this approach is hard to adopt for resource constrained mobile devices. On top of that, having all possible explicit RDF statements in an RDF store is expensive for both updating and querying the data stored in the storage. It is even more expensive for incrementally updating the RDF store because the data needs to be synchronized with the original data sources [15, 17].

To remedy these problems, we propose to create a unified integration view by managing additional graphs to query all personal information desired. Firstly, we manage a "consolidated graph" that contains the aggregated personal information from different data spaces. As illustrated on the left of Figure 2, the consolidated graph provides an aggregated view of personal information, so that a standard SPARQL query processor can provide complete answers relevant to a person. Note that, this consolidated graph uses only one ID scheme that provides a single ID for one person. However, the integration view also has a *SameAs graph* that links consolidated IDs with their counterparts given in other data spaces as shown on the right of Figure 2.

To store and manage the provenance information of the data acquired from difference data spaces, the integration view also stores the data from each data space as a named graph. This enables queries to correlate the consolidated graph with a graph containing information from a particular data space. For instance, the following query is used to query all friends in Facebook that are tagged with "me in a photo" posted in Google+ or Facebook or other data spaces.

```
SELECT ?fbfriend
FROM NAMED ds:facebook
FROM NAMED ds:cg
FROM NAMED ds:sameas
WHERE{
GRAPH ds:facebook{fb:me foaf:knows ?fbfriend}
GRAPH ds:cg{?cgfriend pim:tagged ?photo. ?cgme pim:tagged ?photo.}
GRAPH ds:sameas{?cgfriend owl:sameAs ?fbfriend. ?cgme owl:sameAs fb:me.}}
```

To relieve the user of the burden of using the proper identifiers corresponding to the data spaces and writing such a long query involving the *SameAs* graph, it should be possible to use the user identifiers in queries as all identifiers will be translated to the proper ID scheme based on the context given by the GRAPH keyword. For instance, a Facebook ID *Facebook:me* will be translated to the corresponding one in the consolidated graph by the query processor. The above query could be written in a shorter form as follows:

```
SELECT ?fbfriend
FROM NAMED ds:facebook
FROM NAMED ds:cg
WHERE{
GRAPH ds:facebook{fb:me foaf:knows ?fbfriend.}
GRAPH ds:cg{?cgfriend pim:tagged ?photo. fb:me pim:tagged ?photo.}}
```

To create and maintain the unified view composed from such graphs, we would need a data integration platform that requires several features specifically

designed for mobile devices. The first feature is the data aggregation from heterogeneous data sources. After data is aggregated, it has to be consolidated to create constituent graphs for the integrated view. To store and query data from these graphs, the platform also needs a fully-fledged RDF store tailored to the needs of resource-constrained devices. On top of that, the data in this RDF store has to be accessed in a controlled manner to meet the security and privacy concerns of personal data. These requirements drives the design and implementation decisions of our framework in the following section.

## 3 System design and implementation

To enable querying heterogeneous personal information with the unified view described in previous section, we design the system architecture to meet aforementioned requirements in following. We also describe the implementation of the core component, RDF store for mobile devices, that dictates the expected performance of the whole system in context of resource constraints.

### 3.1 System Architecture

Figure 3 shows the overall system architecture which we will discuss in the following.

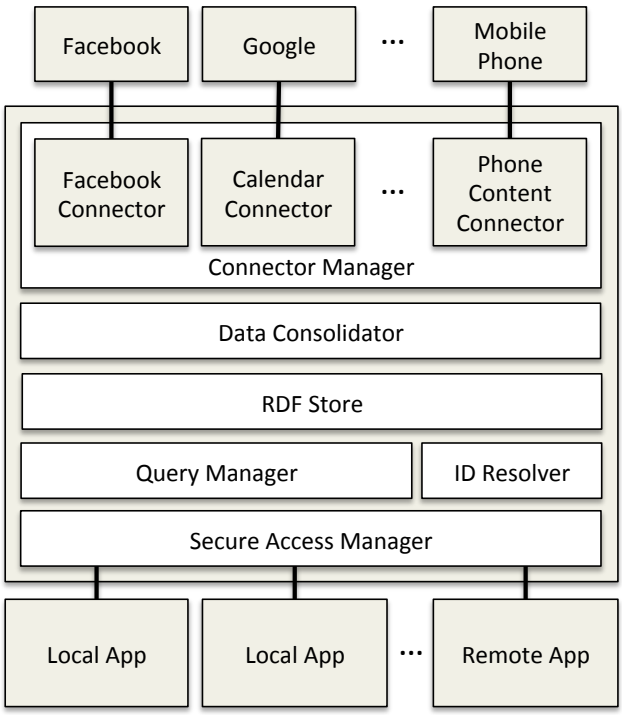



Fig. 3: System architecture overview

As discussed before, there are different sources for personal information like Facebook or Google Calendar that developers should be able to integrate into

our framework. To do so, our framework allows developers to create *Connector* classes and plug them into the framework using the *Connector Manager*. Each Connector is tailored towards a specific information source. So far, we provide a core set of Connectors, namely for Facebook, Google+, Google Calendar,LinkedIn and the local mobile phone content. If developers need to access additional information sources, they can easily implement new connectors for them using the existing ones as a blueprints.

Each connector pulls relevant information from its information source and pushes it towards the *Data Consolidator*. The Data Consolidator consolidates and integrates data from different connectors into the corresponding RDF graphs for each data source. The Data Consolidator also computes the aggregated graph and SameAs graph as described in Section 2. These RDF graphs are stored as an integrated view into the RDF Store.

The *RDF Store* is a core component of our system as it manages triple data directly on the mobile phone instead of on an external server. RDF triples can be stored, indexed and retrieved. The store contains the actual personal data as well as all metadata needed for data consolidation, e.g., user IDs in different data sources and how they relate to each other. The RDF Store has major influence on our system performance and thus must be highly efficient both in terms of execution speed and memory usage. We therefore implemented a RDF store that is specifically tailored towards mobile devices instead of using a feature reduced version of a well known system like Jena. We will discuss the design and implementation of our RDF Store in more detail later in this section.

To access the RDF Store, clients can use two system components, the *Query Manager* and the *ID Resolver*. The Query Manager can handle standard SPARQL queries on the data in the RDF Store. In addition to standard query planning and execution, the Query Manager is also responsible for rewriting queries if necessary. This is the case if the query contains an ID for a user that originates in one of the original data sources, e.g., the ID of a user in Facebook. The Query Manager detects this and rewrites the query such that a consolidated ID is used. This allows clients to place queries without knowing about the data consolidation. From the client's point of view it can use the data as if all of it was available on Facebook. The ID Resolver offers an alternative way of dealing with multiple IDs. It allows a client to request information about a user's ID in different data sources. As an example, a client can ask for the IDs of a user for which it provides the Facebook ID. The ID Resolver looks up the necessary metadata in the RDF Store and returns all IDs for this user, the consolidated ID as well as the user's ID in Google+, etc.

Clearly, security and privacy are major factors when designing a system that manages personal data. Therefore, we chose to add an additional system component, the *Secure Access Manager*, which is responsible for ensuring that all client accesses are done in a secure and privacy-preserving manner. The Secure Access Manager receives requests from local as well as remote client applications. It authenticates the requesting clients and checks their authorisation to access private data. Authorisation is given by the local user using so-called privacy

policies. If access is granted, the Secure Access Manager forwards the request to either the Query Manager or the ID Resolver. It also forwards any results to the requesting client.

To ensure secure communication with remote clients, the Secure Access Manager uses the PIKE approach for secure peer to peer communication establishment for mobile devices [1]. PIKE includes mechanisms to initiate a secure key exchange and to establish a secure network connection with it. It is also able to set up an ad-hoc communication network between mobile devices if necessary.

### 3.2 High performant and low memory consumption RDF store

As presented in Section 2, our solution for maintaining a unified integration view of heterogeneous personal information is to manage a consolidated graph for the aggregated data from different spaces and a sameAs graph to link the consolidated ID with their counterparts. Applying this approach on mobile devices requires a mobile RDF store component which is designed for update-intensive operations. To this end, we built a native and fully-fledged, persistent RDF storage and SPARQL query processor for Android devices, called *RDF On the Go* (RDF-OTG) [3]. RDF-OTG has been extensively used for managing semantic contextual information on mobile devices in PECES [4] and GAMBAS [5] projects. In our implementation, we focused on minimizing the memory footprint and designing data structures tightly coupled with the storage mechanism of mobile devices to achieve maximum efficiency in terms of low memory consumption and high update frequency. In the following we briefly describe our main optimisations to maximise performance and scalability for personal information management applications on mobile devices. A full analysis of the performance gains is given in Section 4.

Reducing memory consumption is one of the critical key targets in mobile DBMS design [10] since most mobile devices have (relatively) limited memory. To achieve that, we reduce the memory footprint of data operations on RDF data by using dictionary encoding, similar to the implementations of JenaTDB or Sesame. Each RDF node is mapped to a compact 32-bit integer with 9 bit to encode the node type and the remaining 23 bit encoding a string identifier which is kept separately on the flash memory instead of in main memory. Most operations on nodes, e.g., matchings during a query execution, can be performed on these node identifiers without accessing the actual string representation. Thus, only one integer must be kept in memory for each node, while string representations can be stored on the flash memory. This leads to a memory footprint of just up to 12 bytes per triple. This is considerably low compared to 450 bytes per triple for the Jena Memory Model as reported by the memory profiler. Note that the compact integer format is used for optimising millions rather than billions of RDF nodes which we believe this is the common scale of most mobile personal information applications. For instance,1.5 million triples are required

[3] RDF-OTG is open-sourced at `https://code.google.com/p/rdfonthego/`
[4] `http://www.nes.uni-due.de/research/projects/peces/`
[5] `http://www.gambas-ict.eu/`

to represent the information of 1200 user profiles (cf., Section 4). However, if necessary, this restriction could be easily removed.

Mobile devices are equipped flash memory as the secondary storage. Flash memory has no mechanical latency, reading is faster than writing and the storage is organized in memory blocks. Instead of reading or writing individual bytes, the I/O unit always reads/writes a whole block. That leads to its erase-before-write limitation when writing a single byte in a block, i.e., the whole block must be read, modified and written again. Thus to achieve the writing requirement, our RDF store needs to optimize writing efficiency rather than reading effiency. To simplify the writing process we use the simplest version of multiple indexing framework of RDF data [4]. It contains only three cyclic orderings of a triple's components with respect to subject(S), predicate(P) and object(O): SPO, POS and OSP. Each indexing order of triples is stored in a separate table.

Due to the impact of flash memory, unmodified versions of traditional data structures do not perform well. On the other hand, flash-aware indexing structure do not work well with "narrow and long" tables as resulting from the above indexing approach. Thus, we use a two-layer indexing approach to manage these tuples of three encoded integers in their corresponding tables. In each table, tuples are sorted lexicographically, partitioned and compressed into individual fixed-size and same-length blocks to the flash I/O block size of the device. The second index layer is a sparse index, small enough to fit into main memory to enable fast lookup for the triples contained in each block. The index holds the lowest and highest node identifier in each sorted block. We also use an in-memory caching mechanism which maintains a limited number of frequently used index blocks.

If a new triple is added, it must be added to the indexes. To do so, the system loads the required index blocks into the cache. Then the triple must be allocated at the right position in the index. This is trivial if the triple should be added at the end of an existing block that still has open space. Otherwise, we would need to move all triples by one position, resulting in a large number of writes. To further reduce the number of read/write accesses, when we need to remove a block from the cache and write it back to flash, our strategy chooses a block that has thehighest chance of not being changed in the future.

## 4 Experimental evaluation

The approach for data integration presented in Section 2 avoids reasoning tasks by modifying RDF triples and then storing them in a unified integration view. This solution is suitable for mobile devices since it does not require much memory for executing the reasoning tasks but it requires a highly performant mobile RDF store when the graphs have to be modified frequently to maintain the unified view. Thus, performance of the system described in Section 3 heavily depends on the performance of the back-end mobile RDF store used. In this section, we present a thorough experimental evaluation[6] of our system's performance and

[6] The description of how to reproduce the results can be found at `https://code.google.com/p/rdfonthego/wiki/SocialNetworkEvaluation`

scalability in terms of data updating and querying. The evaluation uses two system configurations with different mobile RDF stores to evaluate its impact on system's performance and to measure the efficiency gained through our RDF store. In the following we first describe the setup of the experiments and then present and discuss the results obtained from the results.

### 4.1 Evaluation Setup

Our evaluation setup is as follows: To evaluate the impact of our special triple store on system performance, we compare two different system configurations. The first one uses our system as described in the last section, i.e., it uses our triple store, RDF On The Go (RDF-OTG). The original implementation of RDF-OTG was presented in [9]. Since then RDF-OTG has been completely redesigned and reimplemented for maximum performance in Section 3.2. In the experiments we use the most recent version. In the second configuration we replaced RDF-OTG by TDBoid,[7] the Android version of Jena TDB. The rest of the system remained unchanged.

To evaluate how different device profiles with different resources and capabilities impact on the performance, we use three classes of Android devices in the experiments: a HTC desire, a Samsung Galaxy Nexus, and a Nexus 7 Tablet. Their configuration details are described in Table 1.

| HTC desire | Samsung Galaxy Nexus | Nexus 7 Tablet |
|---|---|---|
| AndroidOS 2.3.3 | AndroidOS 4.2.2 | AndroidOS 4.2.2 |
| 998Mhz CPU | 1200Mhz CPU | 1300Mhz CPU |
| 404MB physical RAM | 694MB physical RAM | 974MB physical RAM |
| 32MB DVM heap size | 96MB of DVM heap size | 64MB of DVM heap size |

Table 1: Android devices

For the evaluation dataset, we use a social network data generator [11] to generate three social networks, one for Facebook, one for Google+ and one for LinkedIn. From these we extract relevant data profiles for a person, i.e., the profiles of that person and his/her friends, and feed them into our system. The data generator generates random inverse-identification properties, e.g, mbox_sha1sum, phone from the same dictionary for three social networks so the overlaps are random. With this dataset, we conducted the following four experiments:

*Update throughput:* In the first experiment we tested how much new data the system can incrementally update with a certain underlying RDF store corresponding to each hardware configuration. We simulated the process of data growing by gradually adding more data to the system. We measured the throughput of inserting data (triples/second) until the system crashed or until we reach 1 million triples (whichever happened first).

*Query processor comparison:* In the second experiment we tested the performance and functionalities of TDBoid and RDF-OTG using 8 typical queries with on the maximum data sizes that both TDBoid and RDF-OTG could support. The queries are chosen to cover all query patterns and different complexities.

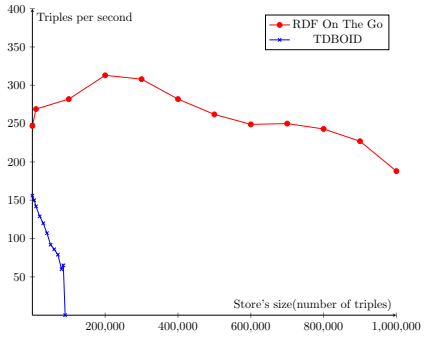
[7] `https://code.google.com/p/androjena/`

Note that, each query accesses to the aggregated view which already involves data from multi-sources and queries 7 and 8 are to show the ability to refer back to the original data sources. The list of the queries in SPARQL language is given in the Appendix.

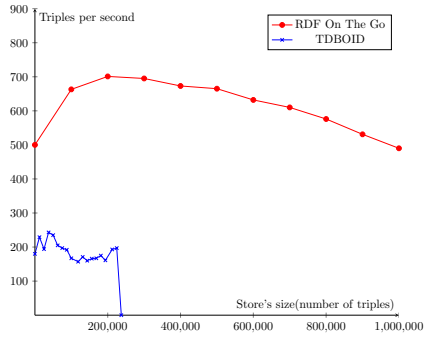
*Memory consumption:* In the third experiment, we measured the memory consumption of two system configurations while performing the queries. The experimental application ran the different queries repeatedly and recorded the maximum memory heap that the operating system allocated for it. To evaluate the impact of the data size on memory consumption, the test was conducted on the Nexus 7 Tablet with five datasets with different sizes. Note that the memory consumption is device-independent. We used the same queries set as in the second experiment.

*Scalability:* In the last experiment we evaluated the scalability of RDF-OTG by measuring the query response time of the above 8 queries on the maximum data sizes that each of the above devices could store.

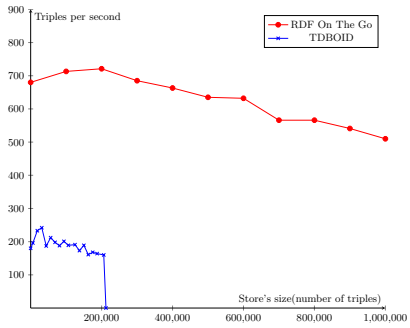
### 4.2 Evaluation Results




(a) HTC Desire




(b) Samsung Galaxy Nexus




(c) Nexus 7 Tablet

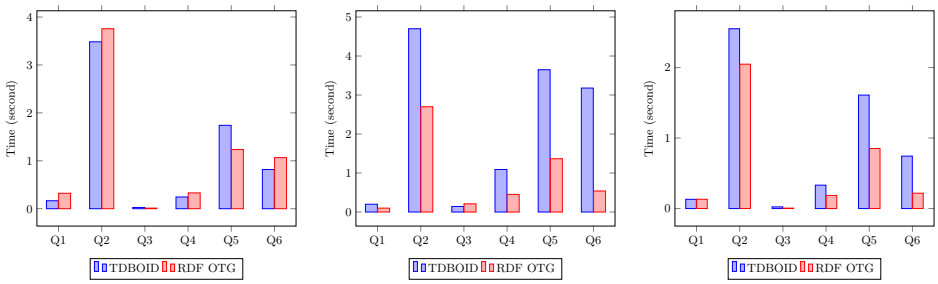
Fig. 4: Updating throughputs

Figure 4 shows the results of our first experiment, in which we measured the update performance of RDF-OTG and TDBoid when adding more and more

triples to the store. As we can see, in general, the writing throughput of RDF-OTG is roughly twice as high as TDBoid's. This shows the advantage of our optimizations for flash memory compared to the design used in TDBoid, which was originally designed for normal magnetic disks.

In addition, while throughput decreases for larger data sizes in the store, RDF-OTG is able to add more triples to the store for all scenarios with acceptable rates (approx. 200 triples/sec for the HTC Desire and approx. 500 triples/sec for the two more powerful devices), even if nearly one million triples were already in the store. TDBoid on the other hand, is not only slower, but it also cannot cope at all with such data sizes and reaches its upper capacity limit at 100k triples on the HTC Desire, 220k triples on the Galaxy Nexus and around 200k triples on the Nexus 7.

In our second experiment we measured the performance of evaluating different queries (as discussed earlier) on existing data sets. The results are shown in Figure 5. Unfortunately, TDBoid does not return any results for Query 7 and 8 because it does not support queries involving named graphs. Therefore, we omitted these two queries from the graphs below. In addition, due to the limitation in the number of triples that TDBoid can handle, we had to reduce the number of profiles contained in the test data for each of the devices: 45 profiles for the HTC Desire, 180 profiles for the Galaxy Nexus and 112 profiles for the Nexus 7.




(a) HTC Desire (b) Samsung Galaxy Nexus (c) Nexus 7 Tablet

Fig. 5: Comparing the query response times of RDF-OTG and TDBoid

The results show that the query performance of RDF-OTG is much higher than TDBoid's for the Galaxy Nexus and the Nexus 7. However, for the HTC Desire, the performance is comparable or even worse for RDF-OTG. The reason for this is that we specifically optimized our system for flash memory. However, the HTC Desire uses an external SD card for storing data instead of internal flash memory. This induces a much higher cost to I/O operations on the HTC Desire. Since TDBoid is originally designed for (relatively slow) magnetic disks, it is able to handle this better than RDF-OTG. However, to do so, TDBoid uses a lot of main memory, which explains its restricted scalability.

The results of the third experiment for measuring the memory consumption for querying are presented in Table 2. Due to the limited scalability of TDBoid exhibited in the inserting throughput test, the tests with TDBoid could only be executed on data sizes of 100,000 and 200,000 triples. The results of the

experiment demonstrate the great improvement in memory footprint optimization of our system. With the same dataset, RDF-OTG requires only one third of the memory that TDBoid needs. For instance, RDF-OTG requires 4MB for the 100,000 triple dataset and 8MB for 200,000 triples to perform the queries while TDBoid requires 11MB and 26MB for the same setup. The efficiency in memory usage also enables RDF-OTG to support much larger datasets. Even with a dataset of 1.5 million triples, the heap size of a system configured with RDF-OTG is lower than 64MB (the JVM maximum heap size of the Nexus 7 tablet).

| | 100k | 200k | 500k | 1m | 1.5m |
|---|---|---|---|---|---|
| RDF-OTG | 4MB | 9MB | 17MB | 34MB | 46MB |
| TDBOID | 11MB | 26MB | N/A | N/A | N/A |

Table 2: Memory consumption of mix queries/size of data

Our last experiment evaluated the scalability of our system. Due to the scalability limitations that we found in earlier experiments, we omitted TDBoid in this experiment and focused on RDF-OTG. Table 3 shows the query response times of the 8 queries on the three devices. As we can see, our system is able to handle datasets of 1 million triples (900 profiles) on the HTC Desire, and 1.5 million triples (1200 profiles) on the Galaxy Nexus and Nexus 7 without any problems. For simple queries like Query 1 and Query 3, it takes less than 1 second to answer the query on datasets of more than one million triples on all devices. More complicated queries such as Query 4, Query 5 and Query 6, take less than 10 seconds, except for Query 5 on the HTC Desire. For this query RDF-OTG crashes with an out of memory error. The HTC's maximum heap size of 32MB is not enough for RDF-OTG to handle the large number of intermediate results generated for this query from the one million triple dataset.[8] We plan to look into this matter further in the future to solve this problem. For the rest of the queries, it takes 10-25 seconds to answer the query. This is due to the time spent for fetching a big set of output results and is determined by the query, so developers have to be careful to "ask the right queries."

| | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q 8 |
|---|---|---|---|---|---|---|---|---|
| HTC (900 Prof/1M tr) | 0.114 | 19.035 | 0.402 | 3.714 | failed | 6.093 | 22.652 | 24.345 |
| GALAXY (1.2K Prof/1.5M tr) | 0.322 | 14.705 | 0.341 | 3.490 | 7.858 | 1.713 | 16.111 | 19.223 |
| NEXUS 7 (1.2K Prof/1.5M tr) | 0.113 | 11.638 | 0.242 | 2.458 | 6.649 | 1.579 | 12.769 | 17.044 |

Table 3: Query response time (seconds) on maximum datasets for RDF-OTG

## 5 Related work

Semantic Web and RDF have long been used as a solution for modeling and integrating heterogeneous personal data. Many works have aimed to better allow a user's access to multiple data silos by using Semantic Web technologies to satisfy the requirements of data portability in terms of identification, personal profiles and friend networks [2]. SemanticLife [8] is one of the early attempts to employ

[8] 34MB would be required as shown in our third experiment.

ontologies for modeling personal digital information. Then there is a series of work on Semantic Desktop such as the Gnowsis Semanic Desktop [14] or the Social Semantic Desktop [5] to provide semantic Personal Information Management (PIM) tools. Additionally, other integrated platforms such as Haystack [12] and Semex [6] provide a wide range of tools and functionalities for PIM. However, they all aim at a standard computer environment and do not take into account mobile devices and their specific problems.

Since then several works have tried to achieve the same functionalities on mobile devices. But the adaptations necessary for the mobile setting proved to be challenging. The first line of work followed the approach of connecting mobile devices to a centralized infrastructure where all processing and storages are delegated to [3, 4]. This line of early work has a lot of security, connection and performance issues. To address them, there are emerging efforts to ship processing and storage of personal information to mobile devices. For instance, [16] tries to store the personal data retrieved from distributed social networks on the phone. However, most of these works still have certain dependencies and use unsecured data exchanges with intermediate parties.

However, these early works have shown the clear interest of using Semantic Web technologies for integrating personal data on mobile devices and the also have shown the need for mobile RDF data processing engines. But these existing works ignore the fact that existing triple storage technologies from normal computers *can not* be directly applied to the mobile setting. For example, the early adoption of Jena to J2ME [7] is micro-Jena [9] which only works on in-memory data on Symbian mobiles. The Android version of Jena, TDBoid, is far better due to newer hardware capabilities but it has a lot of limitations in respect to performance and scalability as we have shown in our experiments in Section 4. We believe this paper is the first to systematically investigate and address the issues of security, integration, performance and scalability of integrating heterogeneous personal information data.

## 6 Conclusions

In this paper, we presented a comprehensive framework for the integration of personal data from heterogeneous data sources, such as different social networks, on mobile devices. Our framework builds upon Linked Data technologies to be generic with respect to the supported data types and data requests, offers a plug-in model to be extensible for additional data sources and relies solely on a user's mobile device, without the need for storing or processing any data on an external, possibly untrusted, server infrastructure. The performance and scalability issues are addressed by our RDF triple store for Android devices, RDF On the Go, which is specifically optimised for mobile devices and flash memory usage. It offers full support for RDF triples and SPARQL queries and is able to handle more than a million triples on typical mobile devices efficiently. Complex queries are supported and can be executed in reasonable time, even for such large data sets but with very small memory footprint.

[9] `http://poseidon.ws.dei.polimi.it/ca/?page_id=59`

## Appendix: Queries used in the experiments

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?per ?property ?info
WHERE {?per foaf:mbox 'mailto:Thierry59@gmx.com'. ?per ?property ?info.}
```

Query 1: Return all information of a person by given mbox

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX db:  <http://dbpedia.org/resource/>
SELECT ?firstname ?lastname ?mbox ?friend ?birthday ?gender
WHERE {
        ?person foaf:based_near db:Bulgaria. ?person foaf:firstName ?firstname.
        ?person foaf:lastName  ?lasttname. ?person foaf:mbox  ?mbox.
        ?person foaf:birthday  ?birthday.?person foaf:gender  ?gender. }
```

Query 2: Extract some informations of people who are nearby Bulgaria

```
PREFIX sibv:   <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX sioc:   <http://rdfs.org/sioc/ns#>
PREFIX dbpo:   <http://dbpedia.org/ontology/>
PREFIX fbp:     <http://www.facebook.com/person/>
SELECT DISTINCT ?location
WHERE { ?user sioc:account_of fbp:p151.
        ?photo sibv:usertag ?user.
        ?photo dbpo:location ?location. }
```

Query 3: Request all the locations that a person has taken a photo

```
SELECT DISTINCT ?properties
WHERE {?person rdf:type foaf:Person. ?subject ?predicate ?person.}
```

Query 4: Request incoming property of a person

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?properties
WHERE {{?person rdf:type foaf:Person. ?subject ?predicate ?person.}
UNION  {?person rdf:type foaf:Person. ?person  ?predicate ?object.}}
```

Query 5: Request incoming and outcoming properties of person

```
PREFIX foaf:        <http://xmlns.com/foaf/0.1/>
PREFIX rdf:         <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sioc:        <http://rdfs.org/sioc/ns#>
PREFIX sibv:        <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX fbp:         <http://www.facebook.com/person/>
SELECT  DISTINCT ?photo
WHERE{
fbp:p39 foaf:knows ?person. ?user sioc:account_of ?person.
?user sibv:like ?photo. ?photo rdf:type sibv:Photo.}
```

Query 6: Request all the photos that are liked by a person

```
PREFIX foaf:        <http://xmlns.com/foaf/0.1/>
PREFIX rdf:         <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sioc:        <http://rdfs.org/sioc/ns#>
PREFIX fbph:       <http://www.facebook.com/photoalbum/>
SELECT  ?per {
GRAPH <facebook> {?per rdf:type foaf:Person.}
GRAPH <master> {?user sioc:account_of ?per. ?user sioc:creator_of fbph:pa103.}}
```

Query 7: Request the photo on Facebook by LinkedIn account

```
PREFIX foaf:        <http://xmlns.com/foaf/0.1/>
PREFIX rdf:         <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sioc:        <http://rdfs.org/sioc/ns#>
PREFIX sibv:        <http://www.ins.cwi.nl/sib/vocabulary/>
SELECT ?user ?c
{
GRAPH <facebook> {?c rdf:type  sibv:Comment. ?user sioc:creator_of ?c.}
GRAPH <master> {?user sioc:account_of <http://linkedin.com/person/p174>}
}
```

Query 8: Request the LinkedIn account of a friend on Facebook

## References


1. W. Apolinarski, M. Handte, M. Iqbal, and P. Marron. Pike: Enabling secure interaction with piggybacked key-exchange. In *PerCom 2013*, 2013.
2. U. Bojars, A. Passant, J. G. Breslin, and S. Decker. Social network and data portability using semantic web technologies. In *SAW 2008*, 2008.
3. M. d'Aquin, A. Nikolov, and E. Motta. Building sparql-enabled applications with android devices. In *ISWC 2011*, 2011.
4. J. David and J. Euzenat. Linked data from your pocket: The android rdfcontentprovider. In *ISWC 2010*, 2010.
5. S. Decker and M. R. Frank. The networked semantic desktop. In *WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, 2004.
6. X. L. Dong and A. Halevy. A platform for personal information management and integration. In *Proceedings of VLDB 2005 PhD Workshop*, 2005.
7. R. B. Hayun. *Java ME on Symbian OS: Inside the Smartphone Model*, volume 30. John Wiley & Sons, 2009.
8. H. H. Hoang, A. Andjomshoaa, and A. M. Tjoa. Towards a new approach for information retrieval in the semanticlife digital memory framework. WI, 2006.
9. D. Le-Phuoc, J. X. Parreira, V. Reynolds, and M. Hauswirth. Rdf on the go: An rdf storage and query processor for mobile devices. In *ISWC 2010*, 2010.
10. A. Nori. Mobile and embedded databases. SIGMOD 2007, 2007.
11. M.-D. Pham, P. Boncz, and O. Erling. S3g2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking*. 2013.
12. D. Quan, D. Huynh, and D. Karger. Haystack: A platform for authoring end user semantic web applications. In *ISWC 2003*, 2003.
13. J. Rekimoto. Timescape: A time machine for the desktop environment. In *CHI 1999*, 1999.
14. L. Sauermann. The gnowsis-using semantic web technologies to build a semantic desktop. Diploma thesis, Technical University of Vienna, 2003.
15. B. Schandl and S. Zander. Adaptive RDF graph replication for Mobile semantic web applications. *Ubiquitous Computing and Communication Journal)*, 2009.
16. S. Tramp, P. Frischmuth, N. Arndt, T. Ermilov, and S. Auer. Weaving a distributed, semantic sociaworkshop on application design, development and implementation issues in the semanticl network for mobile users. In *ESWC 2011*, 2011.
17. G. Tummarello, C. Morbidoni, R. Bachmann-gmr, and O. Erling. Rdfsync: efficient remote synchronization of rdf models. In *ISWC 2007*, 2007.
18. E. R. Weippl, M. Klemen, S. Fenz, A. Ekelhart, and A. M. Tjoa. The semantic desktop: A semantic personal information management system based on rdf and topic maps. In *VLDB 2007*, 2007.