TriplePlace: A flexible triple store for Android with six indices

Natanael Arndt
arndtn@gmail.com

University of Leipzig, Master’s programme in Computer Science

Abstract: This paper presents the idea, design and implementation of a triple store optimized to be used on mobile devices. The triple store implements an indexing structure similar to Hexastore utilizing the Tokyo Cabinet key-value store as persistent storage system. A simple and flexible API gives developers the chance to integrate TriplePlace in their applications to easily use semantic data on Android powered mobile devices. Finally TriplePlace is compared to Androjena in terms of main memory usage, extra storage needed in the final Android Package (APK) and speed.

1 Introduction

More and more data is available in the Linked Open Data cloud [CJ11]. To make this data widely available for personal use, mobile applications become more important. Although mobile hardware begins to have multi core processors and bigger main memory it is still not as powerful as personal computers or even large servers. One method to reduce the load on the mobile end user device would be to store and process the data on a server or in a computing cloud and make the prepared data available through a HTTP interface. But especially mobile devices (e.g. phones and tablets) still don’t have continuous and reliable Internet connection, thus a local data store for offline availability of the data is important.

Those systems which store RDF data [KC04] are often called triple stores. There are already some implementations of triple stores for servers available. A recent benchmark paper [MLAN11] lists the popular triple store implementations Virtuoso, Sesame, Jena-TDB, and BigOWLIM. As of writing this paper only Jena [McB02] is ported to the Android platform in the Androjena project1. There is a note about some parts of the OpenSesame-Project running on Android but no full triple store is available2. On a mobile platform with limited resources, only a small part of the graph can stay in main memory and the data has to be read and written from the persistent storage frequently. Androjena itself has no database integrated for storing the data persistently. Instead, the data has to be serialized and de-serialized to and from RDF/XML [Bec04]. Androjena can be extended

---

1 Androjena: http://code.google.com/p/androjena
2 OpenSesame RDF store on Android: http://weblogs.java.net/blog/fabriziogiudici/archive/2015/07/12/little-teaser-opensesame-rdf-store-android
with TDBoid, the Android version of Jena-TDB\(^3\), to persistently store triples with a higher read and write performance. But to include Androjena and its dependencies in an Android project at least 5.2MB extra JARs are needed, and for TDBoid another 2.8MB have to be added. This results in more than 4MB extra space which is needed for the final Android Package (APK).

Currently the Mobile Social Semantic Web Client (MSSW) [Arn10] uses Androjena to store the WebID of the user and his contacts. In the next release of MSSW we want to use TriplePlace to avoid the drawbacks of Androjena. With TriplePlace we will have a more lightweight triple store which can be adopted to the needs of this project easily.

All system related values in this paper are measured on a Nexus S Phone with Android version 2.3.6. It has 512MB of main memory and a 1GHz CPU.

2 Design

In the following, we describe the design and implementation of TriplePlace. The first paragraph will focus on speed of query processing and shows an approach which gets our result to a level which is comparable to Androjena. In the second paragraph the used persistent storage system is introduced.

Weiss et al. present an approach [WKB08] of a fast answering triple store, Hexastore. It builds an indexing structure which materializes all possible permutations of a Triple \((spo, sop, pso, ...\) in \(3! = 6\) indices. Each index has a head (for spo this is the subject respectively for pso the predicate), a vector (for spo this is the predicate respectively for pso the subject) and a terminal (for spo and pso the object). This gives an advantage on querying the store, because “in a hexastore all first-step pairwise joins are rendered as merge-joins, using the appropriate indices in each case.” [WKB08, p. 6] In addition, Hexastore utilizes a dictionary to map the RDF-Nodes to unique IDs to reduce storage usage. This technique was already successfully implemented on a different mobile platform with limited capacities [WBB08], but since the source code of this project is not freely available we cannot port it to Android. In TriplePlace we adopted some parts of this technique to ensure a fast answering triple store.

In order to minimize the effort of the implementation we are using an existing database management system. Tokyo Cabinet\(^4\) is a space efficient and fast DBM [AT79] which provides hash table (HDB) and B*-Tree (BDB) implementations [10T10]. Because of its usage of mmap() [10T10] it minimizes the file-open-overhead which is very important to achieve a small memory footprint and to speedup reading and writing of triples on a platform with limited main memory.

---

\(^3\)Jena TDB: http://openjena.org/wiki/TDB

\(^4\)Tokyo Cabinet: http://fallabs.com/tokycabinet
3 Implementation

In the following, we first describe some design decisions concerning the persistent storage system, the dictionary and the indexing structure. Later we give an overview of the provided API with its classes and methods.

The Android NDK puts us in the comfortable situation to reuse native (C/C++) code in an Android project. The NDK uses the Java Native Interface (JNI) as interface between native an Java code. Because Tokyo Cabinet already has a JNI-API for Java only the C code had to be compiled with the NDK. The compilation disclosed some minor problems, which had to be solved first. The Android optimized version of Tokyo Cabinet can be found in two git-hub repositories\textsuperscript{5,6}. Tokyo Cabinet is compiled in two shared libraries: libtokyocabinet.so and libjtokyocabinet.so. The Java part of the Tokyo Cabinet Java-API resides in the package tokyocabinet.*.

The dictionary is implemented as hash database (HDB) which is the Tokyo Cabinet implementation of a hash map. It maps the string representation of a RDF-node, as it is used in N-Triples \cite{GB04}, to a long integer (64bit) ID. In order to translate the integer IDs of a query result back to the respective human readable representation an inverse dictionary is implemented. For the hashing algorithm the Java random function is used. It produces a semi-random long integer value, which is used as ID. The new dictionary record is tried to insert in the HDB. If a collision with an existing ID occurs a second ID is generated and if necessary a third and … nth ID. (Note: The ID $0$ is reserved as “ID not specified”) Compared to subsequent insertion with a counter as ID generator the table does not have to be reorganized if the counter reaches its maximum value.

The indexing structure is implemented similar to Hexastore. We are using one BDB, the B$^+$-Tree implementation of Tokyo Cabinet, per permutation. The IDs of the head-, vector-, and terminal-nodes are concatenated and used as key; the value stays empty. This enables us to easily perform a range search with the fixed head and maybe vector and retrieve a list of all matching keys which contains the complete triples.

The TriplePlace API in the package org.aksw.tripleplace consists of following classes:

\begin{description}
\item[Node] represents a RDF-node. The constructor accepts a String which specifies the Node as in N-Triples \cite{GB04}. The following types are supported: literal (e.g. "String"), named resource (e.g. <http://example.com>) and blank node (e.g. _:abc). For the usage in a query pattern the Node can be variable (e.g. ?a). (See also TriplePlace.getNode())
\item[Triple] represents a RDF-statement consisting of subject, predicate and object. The constructor takes three Nodes as parameters. If the Triple is used as pattern also variable Nodes and null are allowed.
\end{description}

\textsuperscript{5}Tokyo Cabinet for Android OS: \url{https://github.com/white-gecko/TokyoCabinet}
\textsuperscript{6}Tokyo Cabinet API for Java on Android OS: \url{https://github.com/white-gecko/TokyoCabinet-Android-API}
**TriplePlace** abstracts the underlying store functions and provides a unified interface. The constructor takes two parameters. The first parameter selects the store engine (currently only `ENGINE_HEXASTORE` is implemented). The second parameter specifies a path in the local file system where the store can save its files.

The `TriplePlace` class defines following methods:

- **getNode()** retrieves the ID of a Node from the Dictionary or adds the Node to the Dictionary and returns a new Node object. It takes a String as parameter to specify the RDF-node similar to the Node constructor. This method should be preferred over the use of the Node constructor, because it already adds the Node to the Dictionary.

- **addTriple()** inserts a Triple to the indexing structure. As argument it takes a Triple object containing three non-variable Nodes.

- **query()** retrieves a list of Triples from the store. It takes a Triple object with fixed and variable Nodes or null-Values. The list will contain all Triples, which match the given pattern.

- **removeTriple()** deletes a Triple from the indexing structure. (Note: this method is not yet implemented)

### 4 Evaluation and Limitations

The implementation of the dictionary hashing algorithm has a complexity of $\mathcal{O}(n)$, where $n$ is the amount of collisions with existing entries. This effectively results in $\mathcal{O}(1)$ without collision, because the probability for a collision is $p = \frac{k}{2^n}$, where $k$ is the number of existing entries in the dictionary. As we see collisions will start to get likely if the dictionary is half full ($k = 2^{64}$) which would take already $2^{64} \cdot 8B = 64\text{EiB}$ only for the IDs which is eight times more than the maximum database size of $8\text{EiB}$ for Tokyo Cabinet [10T10] and would not fit in any mobile device in the near future.

The dictionary encoding reduces the storage consumption of the dictionary and indices from $6 \cdot nB$ to $2 \cdot nB + 6 \cdot 8B^7$. For the minimal HTTP IRI (e.g. `<http://2.am>`) the size of $n = 13B$ which results in a reduction from 78B to 74B per minimal HTTP-IRI-Node. (Note: These values have to be added to the overhead of the underlaying data structure) Because of the usage of the existing key-value store Tokyo Cabinet, it is not possible to use the terminal lists of the indices twice. This increases the needed storage for the six indices in the worst case by a factor of 1.2 for this implementation compared to the original Hexastore. This is six instead of five-fold [WKB08, p. 4] compared to a simple triples table.

TriplePlace needs about a third of the memory compared to Androjena. We have evaluated the memory usage and speed of both systems during the insertion of 2000 Triples.

---

78B is the size of a 64bit integer
drojena allocated about 10000K while TriplePlace needed only about 4000K. TriplePlace was with 5.5s a little slower than Androjena with 4.4s.

Because we are using Tokyo Cabinet through the NDK and JNI there is an overhead on each native function call. This could be reduced by implementing the data structures in Java, but this would be a new task. The usage of Tokyo Cabinet together with the NDK also forces us to use at least API-Level 9 which is currently only installed on 46.3%\(^8\) of the active Android phones.

At the moment TriplePlace supports a rudimentary API for inserting and querying triples. To make it compatible with most of the existing triple stores a SparQL and SparQL/Update query engine and an export to the most common formats RDF/XML, N-Triples, Turtle and Notation 3 [Bec04, GB04, BBL11, BL11] would be necessary. Furthermore named graphs [CBHS05, CS04] are a widely used technique which should be implemented in a following version.

The already high storage overhead of a hexastore could be reduced slightly by using Kyoto Cabinet\(^9\) which has a better performance in terms of storage and also speed compared to Tokyo Cabinet. But it is written in C++ which causes further problems when compiling it with the NDK.

5 Conclusion

In this paper we have successfully demonstrated the idea, design and implementation of a triple store using the Hexastore indexing structure on the Android mobile operating system. The implementation is economically using the limited main memory on mobile devices while it achieves a moderate speed. The simple and flexible API gives other developers the chance to easily use semantic data on Android powered mobile devices. TriplePlace takes less than 1MB of extra storage in the final APK which is a forth of the space needed for the former only solution Androjena.

In the future the TriplePlace query functionality will be extended with the capability of evaluating complex graph patterns instead of currently only simple triple patterns. This will show the currently unused advantages of having six indexes. Furthermore a more complete comparison of the performance of TriplePlace and Androjena with more use cases is necessary.

The source code of TriplePlace is available in a git-hub repository at https://github.com/white-gecko/TriplePlace. It is licensed under the terms of the GNU General Public License (GPL).


\(^9\)Kyoto Cabinet: a straightforward implementation of DBM: http://fallabs.com/kyotocabinet/
Acknowledgments

I would like to thank everybody who supported me and made the completion of this paper possible. Special thanks to my tutor Sebastian Tramp for his inspiration and editorial support, as well as Norman Heino for his support in answering technical questions and Jonas Brekle for the interesting discussions about my ideas.

References


**TriplePlace: A flexible triple store for Android with six indices**

