# Challenge Accepted: QUAD Meets MOCHA2017

Alexander Potocki(✉), Daniel Hladky, and Martin Voigt

Ontos GmbH, Leipzig, Germany
{alexander.potocki,daniel.hladky,martin.voigt}@ontos.com

**Abstract.** Native RDF (http://www.w3.org/RDF/) stores have been making enormous progress in closing the performance gap compared to relational database management systems (RDBMS). But this small gap, however, still prevents the adoption of RDF stores in scenarios for large-scale enterprise applications. We solve this problem with our native RDF store QUAD and its fundamental design principles. It is based on a vector database schema for quadruples and it is realized by facilitating various index data structures. QUAD also comprises approaches to optimize the SPARQL query execution plan by using heuristic transformations. In this short paper, we briefly introduce QUAD and sketch in which tasks of the Mighty Storage Challenge we will attend to benchmark the current performance capabilities.

**Keywords:** RDF · SPARQL · Index · Query optimization · Benchmarking

## 1 Introduction

Over the past few years, we have seen explosive growth in the dissemination and use of semantic data. Initially, the traditional application fields of semantic technologies were areas as medicine, bioinformatics, public administration with their Linked Open Data portals. For the further establishment in other domains on enterprise-scale reliable and efficient solutions for storing and querying permanently increasing volumes of semantic data are the main foundation. Here, the *Mighty Storage Challenge* will contribute to a big extent.

Our goal is to provide an universal and customizable solution for storing semantic data that is efficient concerning its performance, does not require the use of a relational DB and translation of SPARQL into SQL. It needs to support recommendations of the World Wide Web Consortium (W3C) as RDF, SPARQL 1.1 [2], and SPARQL protocol[1]. Our RDF store **QUAD** [1] is the result of our ongoing research and development. In this introductive paper, we briefly describe our QUAD and explain how we are participating in the *Mighty Storage Challenge* to benchmark and compare the performance of our solution to other available stores.

[1] http://www.w3.org/standards/semanticweb/.

© Springer International Publishing AG 2017
M. Dragoni et al. (Eds.): SemWebEval 2017, CCIS 769, pp. 16–20, 2017.
https://doi.org/10.1007/978-3-319-69146-6_2

## 2 Related Works

Developing an RDF database, every developer faces some challenges. The two main problems are the following.

First, the choice, conceptualization, and development of a complete index set on SPO[2] triples or SPOG (see footnote 2) quadruples are complex. A series of works by Harth et al. [3–5], Baolin et al. [6], Weiss et al. in [7], and Abadi et al. [8] demonstrate similar methods of constructing them, using prefix search, reduce the full set of indices. Pursuing the elaboration of the multiple-index approach proposed in [3,9] and improved in the Hexastore solution [7], we have created a database structure supporting certain permutations of a set of elements for quads within SPOG (see footnote 2) relations.

The second challenge is the conceptualization and development of a query execution plan (QEP [14]). Several researchers formerly addressed the fundamental issues. Neumann et al. [10,11] proposed the query optimizer which mostly focuses on join order when generating execution plans and uses dynamic programming for the plan enumeration, with a cost model based on RDF-specific statistical synopses. Stocker et al. [12] presented SPARQL query optimization methods based on static optimization using the Basic Graph Pattern (BGP) method to optimize triple pattern join sequences. Gomathi et al. [13] described a multi-query optimization process that splits an input query into clusters using the K-means method based on the common sub-expression in the queries constituting an input set of queries. During our research, we examined equivalent query plan transformations based on heuristic rules worked out using computational complexities of algorithms for implementing operations, experimentally as well as through the observation of the system response times for various QEP configurations. This research direction was the most promising for us so that we have developed a set of heuristics never published in other works. We employ them for the static optimization of the original QEP.

## 3 QUAD: Design and Implementation

QUAD follows the generally accepted design of databases, which is sketched in Fig. 1. In order to receive and process queries from client applications, QUAD implements *SPARQL 1.1 Protocol*[3]. Before making a request to QUAD, any client application must authenticate itself using *Digest Access Authentication*[4] protocol. After the authentication process, QUAD creates a client session object and associates an access descriptor or, so called, authorization token with it, which determines the availability of particular data for subsequent client queries. Only then SPARQL queries are ready to be parsed and converted to the iterator tree in the *SPARQL Engine* module. The authorization token is accounted for by placing additional filter iterators in the tree. Leaf iterators are connected to

[2] Here, “S” stands for Subject, “P” for Predicate, “O” for Object, and “G” for Graph.
[3] https://www.w3.org/TR/sparql11-protocol/.
[4] https://en.wikipedia.org/wiki/Digest_access_authentication.

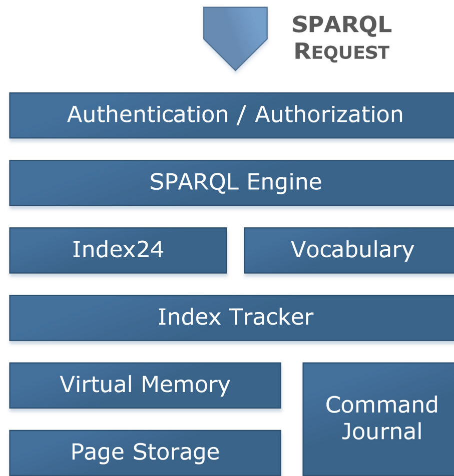


**Fig. 1.** Architectural design of QUAD.

a set of indexes that store different slices of the RDF data (*Index24*), as well as to the indexes of the literal values (*Vocabulary*). Indexes are implemented using the BTree algorithm. The nodes of the BTree are represented by blocks of memory or pages of given size specified during the configuration of the QUAD database. Each block has its unique numeric identifier. The *Virtual Memory* subsystem provides access to the pages by their identifiers and also caches them using the *2Q buffer cache algorithm* (Johnson et al. [15]). The *Page Storage* subsystem is responsible for loading and uploading data to permanent storage. This subsystem uses direct access to storage devices, bypassing the operating system's file cache to maximize performance. The *Index Tracker* tracks any changes to pages during the insertion, deletion or modification of data in BTree indexes. These changes are encoded by a set of incremental instructions, which in turn are stored on permanent storage by the *Command Journal* subsystem. These records, called the transaction log, can be used to restore the database in the event of an emergency shutdown.

As Fig. 1 illustrates, QUAD follows a component-based database design. Each component is described by its interface. The implementation details of the component are hidden from the other ones. Instances of components have unique identifiers. These identifiers serve to bind them to each other. The component life-cycle and their binding are managed by a specially developed framework that implements naming services, state storage services, and configuration services. QUAD is implemented in C++11 with intensive use of generic programming techniques. The architecture and operation system abstraction layer is performed mainly using the Boost library[5]. Assembly and testing were carried out on Linux

[5] http://www.boost.org/.

operating systems (Ubuntu and CentOS), Windows and Android, X86-64 and ARMv06 platforms.

## 4 Evaluation

To evaluate the performance of QUAD and compare it with other well-known RDF storages, we are going to participate in the *MIGHTY STORAGE CHALLENGE competition - ESWC 2017*[6]. This competition offers four types of tasks, for a comprehensive assessment of the performance of RDF storage. 1, 2 and 4 of these tasks are the general tests of intensive loading of RDF data in parallel mode and query execution over this data. These test scenarios emulate the database operating modes in real business tasks. The third task is related to the evaluation of the efficiency of storing versioned RDF data. QUAD does not contain any particular versioning implementations, so we can only emulate versioning using named graphs for different versions of RDF data. Since this approach is not efficient and may only offer a baseline performance, QUAD does not challenge this task.

For the competition, we prepared a special version of QUAD, configured and packed it into a docker image. The contest does not involve data durability testing, so we've disabled transaction journaling. For non-blocking data reads during the write operations, we activated the MVCC[7]. Almost all RAM is used for the index page cache. The number of threads executing simultaneous requests to the database corresponds to the number of processor cores in the system.

## 5 Conclusion and Further Work

In this introductive paper, we give a brief overview of our RDF store QUAD for the interested readers of the MOCHA2017 papers. If our native RDF store fits the challenge requirements, we are looking forward to the invitation to the tasks 1, 2 and 4 in order benchmark the already prepared dockerized version.

Besides the challenge, our ongoing work is to add features, stabilize them and boost the overall performance of QUAD. Regarding the latter, our primary focus is the development of a RDF data store cluster, which is geared towards the multi-platform processing of very-large-scale RDF datasets larger than 1 billions of triples. Therefore, we facilitate the concept of the parallel deployment of independent, full-featured RDF stores instance with a shared vocabulary index. Such an approach will prohibit the multiple storages of the same literal values in different stores, as well as to have a unique identification of RDF entities across all RDF stores in the cluster. One of the principal challenges in building a distributed database is QEP planning. Delays in transferring data between hosts can significantly reduce query performance. Hence, we developed a particular statistical index, which radically reduces the amount of data sent.

[6] https://project-hobbit.eu/challenges/mighty-storage-challenge.

[7] https://en.wikipedia.org/wiki/Multiversion_concurrency_control.

**Acknowledgments.** This work was partially supported by the BMWi project SAKE (Grant No. 01MD15006).

## References

1. Potocki, A., Polukhin, A., Drobyazko, G., Hladky, D., Klintsov, V., Unbehauen, J.: OntoQuad: native high-speed RDF DBMS for semantic web. In: Klinov, P., Mouromtsev, D. (eds.) KESW 2013. CCIS, vol. 394, pp. 117–131. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41360-5_10
2. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Technical report, W3C Recommendation (2013). https://www.w3.org/TR/sparql11-query/
3. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: LA-WEB (Latin American Web Congress) (2005)
4. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: a federated repository for querying graph structured data from the web. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007). doi:10.1007/978-3-540-76298-0_16
5. Harth, A., Decker, S.: Yet Another RDF Store: Perfect Index Structures for Storing Semantic Web Data With Context, DERI Technical report (2004)
6. Baolin, L., Bo, H.: HPRD: a high performance RDF database. In: Li, K., Jesshope, C., Jin, H., Gaudiot, J.-L. (eds.) NPC 2007. LNCS, vol. 4672, pp. 364–374. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74784-0_37
7. Weiss, C., Karras, P., Bernstein, A.: Sextuple Indexing for Semantic Web Data Management. PVLDB **1**(1), 1008–1019 (2008)
8. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: VLDB, pp. 411–422 (2007)
9. Wood, D., Gearon, P., Adams, T.: Kowari: a platform for semantic web storage and analysis. In: XTeGh (2005)
10. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. J. VLDB **19**(1), 91–113 (2010)
11. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB **1**(1), 647–659 (2008)
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: WWW 2008, pp. 595–604. ACM, New York (2008)
13. Gomathi, R., Sathya, C.: Efficient optimization of multiple SPARQL queries. IOSR J. Comput. Eng. (IOSR-JCE) **8**(6) (2013), pp. 97–101 (2013). www.iosrjournals.org, e-ISSN: 2278–0661, p- ISSN: 2278–8727
14. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
15. Johnson, T., Shasha, T.: 2Q: A low overhead high performance buffer management replacement algorithm. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994), San Francisco, CA, USA, pp. 439–450 (1994)