OntoQuad: Native High-Speed RDF DBMS for Semantic Web

Alexander Potocki[1], Anton Polukhin[1], Grigory Drobyazko[2], Daniel Hladky[2], Victor Klintsov[2], and Jörg Unbehauen[3]

[1] Eventos, Moscow, Russia
{alexander.potocki,anton.polukhin}@my-eventos.com
[2] National Research University - Higher School of Economics (NRU HSE), Moscow, Russia,
{gdrobyazko,vklintsov,daniel.hladky}@hse.ru
[3] Universität Leipzig, Institut für Informatik, Leipzig, Germany,
unbehauen@informatik.uni-leipzig.de

Abstract. In the last years native RDF stores made enormous progress in closing the performance gap compared to RDBMS. This albeit smaller gap, however, still prevents adoption of RDF stores in scenarios with high requirements on responsiveness. We try to bridge the gap and present a native RDF store "OntoQuad" and its fundamental design principles. Basing on previous researches, we develop a vector database schema for quadruples, its realization on index data structures, and ways to effectively implement the joining of two and more data sets simultaneously. We also offer approaches to optimizing the SPARQL query execution plan which is based on its heuristic transformations. The query performance efficiency is checked and proved on BSBM tests. The study results can be taken into consideration during the development of RDF DBMS's suitable for storing large volumes of Semantic Web data, as well as for the creation of large-scale repositories of semantic data.

Keywords: RDF, SPARQL, index, multiple joins, query optimization

1 Introduction

The *research goals* addressed in this article are: creation of a native RDF DBMS that is efficient in terms of its performance, does not require the use of a relational DB and translation of SPARQL into SQL, and supports such recommendations of the World Wide Web Consortium (W3C) as RDF, SPARQL 1.1 [1], SPARQL protocol[1]; elaboration on existing approaches to the creation of a native RDF DBMS information architecture; development of new heuristic algorithms for optimizing query execution plans. An RDF data representation model is one of the foundations of the Semantic Web concept developed and being promoted by the W3C. Accordingly, an RDF DBMS is one of its fundamental underlying tools. The efficiency of any given RDF DBMS depends on its capabilities tailored

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

to a specific range of tasks. These are, first of all, volumes of processed data, and performance, i.e., the ability to efficiently execute advanced queries on such data. Although there are already RDF DBMS's designed to process voluminous data (hundreds and thousands of billions triples), a task of searching and developing optimal storage structures and RDF data processing algorithms continues to be relevant. *The main contributions* of this paper are: (a) evolvement of a vector model for representing triples into a vector model for representing quadruples, and its efficient implementation by means of index structures for quadruple data storage and retrieval; (b) presentation of the *multiple join* query execution plan operator implemented by the *zig-zag join* method for joining several data sets simultaneously; (c) a set of heuristics for the static optimization of SPARQL queries. The conducted work has resulted in the practical embodiment of (a), (b) and (c) as an efficient native RDF DBMS "OntoQuad" that can be used for creating large-scale Semantic Web data repositories.

2 Related Works

In a series of works by Harth et al. [2,3,4] focused on the YARS2 system development there is a discussion of issues related to the building of a complete index on SPOC[2] quadruples, or quads, thus enabling direct lookups by multiple dimensions with no need for joins. The total number of quad indices is determined, such indices being necessary for performing the full search by all possible complete quadruple patterns (16 patterns, 16 complete quad indices) which is then reduced to six indices with an alternative ordering on the assumption that the indices can be used to support prefix search that enables lookups with a partial key in case of incomplete quads. A problem of choosing index structure variants for the implementation of selected indices is investigated, and a sparse index variant is proposed. A similar method is presented by Baolin et al. in [5] where, proceeding from all possible SPO[2] triple patterns, a three-index system is proposed ensuring the prefix lookups. Weiss et al. in [6] presented an approach and evaluation of a prototype of one of the column-oriented vertical-partitioning (COVP) method variants. In this approach the fundamentals of vertical partitioning (refer to Abadi et al. [7]) and multi-indexing (refer to Wood et al. [8] and Harth et al. [2]) systems are taken. An RDF triple of the SPO type is indexed within a multiple-index construct that associates two value vectors with every RDF element, one vector per each of the two residual RDF triple elements. A six-index structure is proposed for indexing RDF data, these indices being used to materialize any orders of precedence consisting of three SPO-type RDF elements. Pursuing the elaboration of the multiple-index approach proposed in [2,8] and improved in the Hexastore solution [6], we have created a database structure supporting certain permutations of a set of elements not only for SPO triples, but also for quads belonging within SPOG relations. We develop a vector model of the Hexastore to represent triples, and extend it onto the quadruple

[2] SPOC stands for Subject Predict Object Context, and SPO stands for Subject Predict Object

representation. The model we propose consists of four levels corresponding to the positions of elements in a quad; upon each of these levels a vector of element values conforming to a quad element value at a previous level is generated. The next important problem we solved in the course of creating the OntoQuad RDF DBMS was a problem of generating a query execution plan that would ensure a query execution time gain in comparison to an initial performance plan produced by the SPARQL parser. Query execution plan optimization issues were formerly addressed by several researchers. Neumann et al. [9,10] proposed the query optimizer which mostly focuses on join order in its generation of execution plans. It employs dynamic programming for plan enumeration, with a cost model based on RDF-specific statistical synopses. Stocker et al. [11] offered SPARQL query optimization methods based on static optimization with the use of the Basic Graph Pattern (BGP) triple pattern join sequence optimization. The query execution plan optimization is performed before its efficiency is evaluated, and is achieved by means of the triple-pattern join selectivity estimation heuristics along with generalized statistics on RDF data to support those heuristics. In Hartig et al. [12] the SPARQL Query Graph Model (SQGM) was proposed supporting all phases of the query execution process. The optimization consists in transforming the initial SQGM into a semantically equivalent one to achieve a better execution strategy. Such transformation is performed by means of a set of the Rewrite Rules included in the heuristics on transforming the initial SQGM into a resulting model, which ensures the achievement of a specific query transformation goal. Pérez et al. [13] presented an approach to formalize the semantics of the core of SPARQL. The work defines a set of equivalence expressions that allow transforming any SPARQL query into a simple normal form consisting of unions of graph patterns. Gomathi et al. [14] described multi-query optimization process which is based on input query partitioning into clusters using K-means clustering based on the common sub-expression in the queries that are provided as Input. During our research we examined equivalent query plan transformations based on heuristic rules worked out using computational complexities of algorithms for implementing operations, experimentally and through the observation of the system response time for various query execution plan configurations. Proceeding in this direction, we developed a set of heuristics never referred to in the works examined by us, and used it for the static optimization of the initial query execution plan.

3 Data Base Organization

Definition 1. *Under a given set R of URI hyperlinks, set B of empty nodes and set L of literals, a triple* $(s, p, o) \in (R \cup B) \cup R \cup (R \cup B \cup L)$ *is named an "RDF triple". A pair (t, c) with a triple t and* $c \in (R \cup B)$ *is named a "triple in context c". A triple ((s, p, o), c) in context c is treated as a quadruple (s, p, o, c) [2].*

In the (s, p, o) triple, s is named a "subject", p is named a "predicate" or "attribute", o is named an "object". A concept of context is introduced to trace the origin of a triple in an aggregating graph G.

In our work we elaborate on the vector representation of triples proposed for the Hexastore [6], by expanding it onto quadruple representation. This model consists of four levels corresponding to the positions of elements in a quad; upon each of these levels a vector of such element values is generated. The root level (level 1) contains a vector of the values of four elements S, P, O, G. Level 2 of the model contains groups of vectors, where each group has one of the level 1 element values as its parent. For every element of one of the four types admissible at the first level there exist three vectors consisting of values corresponding to three admissible residual elements of a quadruple (e.g., if an instance of element P is chosen at level 1, then vectors of values of the S, O and G-type elements will conform to it at level 2). Two vectors of values corresponding to the two admissible residual elements of a quadruple will conform to an element value at the second level (e.g., if an instance of element P is chosen at level 1, and an instance of element S is chosen at level 2, then vectors of values of the O and G-type elements will conform to it at level 3).

The DBMS creates several files for storing data, which correspond to four levels of a quadruple representation logical structure. Each file is meant for storing quadruple element values belonging to only one of the levels with a specified sequence order of such values. The file combines within itself both a structure for storing data and an index structure implemented as *B-trees* in the form of the Key-Value ordered lists. We employ *B-tree* as a constant index data structure because it ensures the support of prefix lookups by key value ranges, and one triple index based on *B-tree* can be used for queries by several triple patterns.

Each file contains a structure that is a clustered relation consisting, subject to a level, of either two (for level 1 and level 4) or three (for level 2 and level 3) fields. The *B-tree* index being created is a clustering index ([15]), and its search key defines the ordering of the file entries, which ensures the efficient performance of join operations. For the first-level vectors corresponding to the S, P, O and G quadruple elements, four separate files are created, one per each of the said elements. For the second and subsequent levels, vectors of values corresponding to the S, P and G elements are placed into one file, while vectors corresponding to the O element are placed into another separate file. Each file has its own *B-tree* index independent of other files' indices, which makes it possible to perform both index search within every single segment at each level, as well as simultaneous joining by several linked levels. Fig. 1 shows a fragment of a quadruple index structure. Let us inspect it closer.

At level 1 the key is a reference to an attribute value from an S, P, O or G domain stored in the system *Vocabulary*. For each value of the first-level key belonging to one of the elements there are three unique auto-incremental numbers serving as indicators of three possible quadruple elements that correspond at the second level to an element of the first level. For example, for each instance

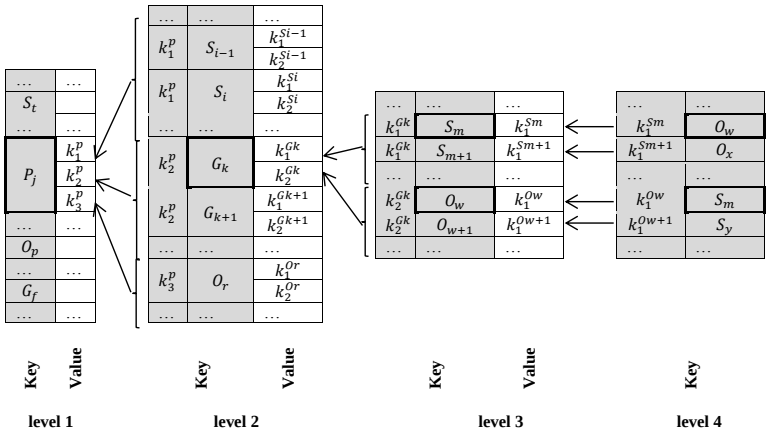

Fig. 1. Quadruples index structure fragment

belonging to element P there will be corresponding unique numbers designating elements S, G and O, which for the instance P_j in Fig. 1 designated as k_1^p, k_2^p and k_3^p.

At the second level a key of an element instance is an object consisting of two fields: reference to the value of a "parental" instance residing at the first level, and reference to an attribute value stored in the system *Vocabulary*. For each value of the second-level element instance key belonging to one of the admissible quadruple elements there are two unique auto-incremental numbers serving as indicators of two possible quadruple elements that correspond at the third level to an element instance of the second level. For example, for each instance belonging to element G that refers to the first-level instance of element P there will be corresponding unique numbers designating elements O and S, which for the instance G_k in Fig. 1 are designated as $k_1^G k$ and $k_2^G k$. And so on, through to the fourth level.

Our data storage structure is a variant of a method that implies the vertical partitioning of the complete relation of SPOG quadruple instances into subsets, offered by Abadi et al. [7] in order to increase the efficiency of system search query responses. As distinct from [7], where it is proposed to create separate tables for every predicate P with two columns S and O, each of the four SPOG elements in our schema is a peer entity, that is why a quadruple is partitioned vertically (by levels) in accordance with the approach described above, which makes it possible to efficiently perform search not just in case of a known predicate, but for any S, P, O or G element. Such a structure resembles the data organization in a column-oriented DBMS focused on operations with columns [16]. A dissimilarity of the data storage schema under consideration from the column-oriented DBMS schema lies in a method of binding data belonging to the same tuple. While a traditional columnar storage schema implies, for every data element instance, the presence of a pointer to a tuple to which this element belongs, such a pointer in our schema points at an element preceding the selected element in the quadruple tuple. Because of the fact that the columns contain not values themselves, but

links to *Vocabulary* items, thus making it possible to take advantage of columnar database features and efficiently compress files on disks and in RAM [17].

Vocabulary is a comprehensive lexicon of URI's and literals that are "known" to the base which associates the values of S, P, O and G with their vocabulary ID's being unique within a DB instance. By introducing the *Vocabulary* we achieve: (a) speed-up of processing the types instances of which are placed in the *Vocabulary* during data write-read operations; (b) reduction of space occupied by DB index files on HDD and in memory due to the fact that a long value is stored in the *Vocabulary* in a unique copy, while indices use a short numeral reference to it. That way, we implemented a vector model for storing data that was initially proposed for triples in the Hexastore and afterwards expanded by us onto the quadruple data storage schema focused on the column-wise storage of quad-tuple data at the level of physical storage organization. Our storage schema implementation allows for the high performance of DBMS, as, due to such indexing method, the work of leaf *iterators*, supporting data retrieval, lookup and joining operations will always be performed on ranked data, which guarantees the logarithmic (based on a number of DB entries) query processing time.

4 Architecture and Implementation

4.1 DBMS Components

The main system components and their interaction when executing queries described below. Here, the *SPARQL parser* is a unit responsible for the syntactic analysis of queries and generation of the initial query execution plan tree [19]. The tree nodes contain procedures/operators (hereinafter, operators) implementing operations of relational algebra [15], the majority of which correspond to SPARQL algebra operators [1]. Directional links between the nodes (direction from leaves to the root) signify the operators' arguments. The main arguments of leaf operators are quad patterns and index files of the base. Higher-level operators in the execution plan tree may have one (e.g., *Order by* or *Distinct*), two (e.g., such operators as *join*, *Cartesian product*) or more (as in the *multiple join*) lower-level arguments/operators. In our system the operators are implemented by *iterators* [19]. *Iterator* is an object the interface of which includes such methods as *empty()* – check if the data set is empty, *next()* – go to the next dataset record, *lowerBound()* – logarithmic complexity search by ordered data, *setRange()* – logarithmic complexity search of value ranges. The query execution plan tree is processed in the following way. To obtain the SPARQL operator execution result, the *next()* method of the root iterator is called, upon which the root iterator will call the *next()* method of the next iterator occupying a lower place in the stack of *iterators*. The process will go on until the leaf *iterators* working with database index files are reached. The *Optimizer* is responsible for transforming the initial query execution plan into a new equivalent plan, more optimal in terms of performance time and resources. The *Optimizer* can choose one or another algorithm for instantiating relational algebra operators. When constructing a plan, the *Optimizer* employs knowledge about computational

complexity and resource intensity of the *iterators*, as well as the *Vocabulary* and *indices*. The *iterators* and *Optimizer* interact with the *indices* (*Index-24 Storage*) and *Vocabulary*. *Functions* are either functions of the SPARQL language (e.g., *coalesce*, *if*, *sameTerm* and others [1]) or other custom functions. The *functions* are used by the *iterators* and retrieve variable values from the *Vocabulary*. The instantiated *indices* and *Vocabulary* store their objects in the *Database File Storage*, in a *B-tree*. An intermediate level between the *File Storage* and system of *indices* (*Index-24 Storage*)/*Vocabulary* is a cache of database file pages (*Database Cache*).

4.2 Datasets Join Iterators

We distinguish two types of iterators[3] responsible for joining datasets. These are the iterator family *natural join* $\bowtie\ (L, R)$ [15] (hereinafter, *join*) performing the join of two (left and right) sets, and the iterator family *multiple join* $\bowtie_M\ (R_1, R_2, ..., R_n)$, performing the join of several sets simultaneously. Besides, judging by a sorting feature of datasets being joined, we distinguish iterators with sorted and unsorted arguments. Thus, the iterator family *join* includes the following iterators: $\bowtie\ (sorted, sorted)$, $\bowtie\ (unsorted, sorted)$, $\bowtie\ (unsorted, unsorted)$. To employ the *multiple join* iterators, the sorted dataset condition must be fulfilled for all arguments except, perhaps, for the first one: $\bowtie_M\ (sorted, sorted, ..., sorted)$ and $\bowtie_M\ (unsorted, sorted, ..., sorted)$. Each iterator has two pointers to a current position in a dataset, *begin* and *end*, which can be set using three techniques. The first technique consists in employing the *lowerBound(key)* iterator method that sets a position of the *begin* pointer indicating the beginning of an ordered relation subset every tuple of which has a key fragment greater than or equal to a specified key value. The *end* is considered identical to a pointer to the end of the range, installed by an iterator located on the level above. The second technique consists in employing the *setRange(key)* iterator's method and makes it possible to set a scanning range [*begin*, *end*] as a value of a function of the searched key. As practice shows, the *lowerBound* method is faster than the *setRange* method because it requires a less number of disk reads (only the upper boundary of the key needs to be found in a dataset, while the *setRange* method implies the search of both upper and lower boundaries). The third technique, or the *next()* iterator method, shifts a position of the *begin* pointer by an entry next to the current one. In case of leaf operators, the *lowerBound(key)*, *setRange(key)* and *next()* methods work with *B-tree* indices.

4.3 Join of Two Datasets

Let us explore the $\bowtie\ (sorted, sorted)$ operator as exemplified by joining two patterns, $?s : p_1\ ?o_1\ ?g_1$ and $?s : p_2\ ?o_2 : g_2$ and demonstrate the principles of such join technique. Here $: p_1$, $: p_2$, $: g_2$ – are pattern constants, and $?s$ is a variable by which the join of datasets is performed. In case of such join, the

[3] iterators below are written in prefix notation

transformation of patterns is performed at first, when constants in every pattern are "shifted" to the left and variables by which the join is executed are placed immediately following the constants: : p_1 $?s$ $?o_1$ $?g_1$ and : p_2 : g_2 $?s$ $?o_2$. For each pattern from the set of available index structures such a structure is chosen the key components of which in the left-to-right order correspond as fully as possible to the pattern constants. Thus, the PSOG and PGSO index files will be chosen. Then one of the leaf iterators (*indexScan*) is selected as a left (L) argument of join $\bowtie$, while the other iterator becomes a right (R) argument: $\bowtie_{s1=s2}$ $(indexScan(PGSO, key_L), indexScan(PSOG, key_R)$. It is worth to choose such iterator in the capacity of L for the dataset generated by the triple pattern of which the condition $|L|<|R|$ is fulfilled, where $|L|$ and $|R|$ are dataset dimensions. During the index scanning the keys key_L and key_R are generated. The beginning part of such key is formed by inserting constants from query patterns for L and R. These keys are then used by the leaf iterators for obtaining L and R dataset entries being determined by specified patterns. The join is performed in the following way. A process $\bowtie$ generates key_L from the pattern constants and the join variables and initiates a call to the $lowerBound(key_L)$ method of the leaf iterator $indexScan_L(key_L)$, which sets the scanning range [*begin*, *end*] in L and, by means of the *next()* method, obtains the next tuple that is transferred to $\bowtie$. In $\bowtie$ a current value of the join variable $?s$ is extracted from the "left" tuple. This value participates in generating the key_R key to perform lookups in R. Then the iterator $\bowtie$ calls the $lowerBound(key_R)$ method of the leaf iterator $indexScan_R(key_R)$, which, similar to the left index scan, sets the scanning range [*begin*, *end*] in R and performs the index search of tuples with the key_R key. The ordering of R by join variables guarantees that, if such a range exists, then a join variable value (*s2*) from the first tuple of this range will be $\geq$ the last specified join variable value (*s1*) from L. If the scanning range [*begin*, *end*] for R is found, then, using the *next()* method, all tuples will be extracted from it one-by-one and returned to $\bowtie$. Current values of the join variables will be extracted from every such tuple, and if the join variable values of tuples "on the left" and "on the right" are identical, the iterator $\bowtie$ performs a join of a tuple having come from L with tuples having come from R. As soon as the tuples from the "left" or "right" range [*begin*, *end*] become exhausted, the first out-of-this-range tuple is chosen from a dataset currently being scanned (it can be either L or R), upon which in $\bowtie$ a current value of the join variable $?s$ is extracted from that tuple and becomes involved in the process of generating a key for lookups in an opposite set. The process is repeated from the moment of setting a scanning range by means of the $lowerBound(key_i)$ method in an opposite dataset. In such a manner, the algorithm alternatively moves between entries of the left relation L and entries of the right relation R, resulting in the formation of a join of the left and right tuples provided that their values coincide in the positions specified by the join variables. Fig. 2 schematically shows the process of joining two ordered sets by variable S.

Judging by the look of the left/right dataset alternating scanning trajectory, the algorithm we employ belongs to the *zig-zag join* type [15,18]. It allows us

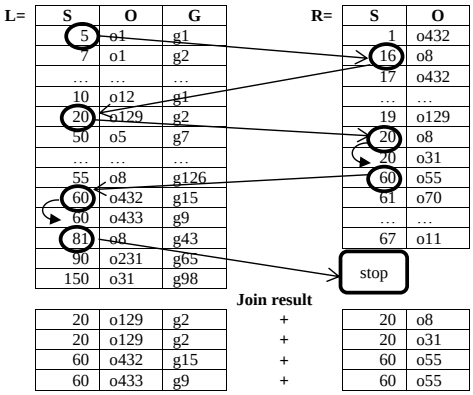

Fig. 2. Quadruples index structure fragment

to avoid lookups (and disk readouts) of "unnecessary" tuples from the left and right sets, which is a common thing for the join and index based join algorithms [15], as well as does not require any re-sorting of the left and right sets in case of leaf iterators because a properly chosen *B-tree* index guarantees that these datasets are properly sorted. The join algorithm terminates when, at the next $lowerBound(key_i)$ lookup attempt, a returned subset of tuples is empty. In case of the $\bowtie$ $(unsorted, sorted)$ join, the whole left set can be scanned consistently, entry by entry, instead of setting a lower boundary by the left set $lowerBound(key_L)$.

4.4 Multiple Join of Several Datasets

The above-presented *zig-zag join* algorithm for two L and R sets we extended onto the *multiple join* $\bowtie_M (L_1, R_1, R_2, ..., R_n)$ operation of joining several sets simultaneously. Such a join is performed in a cascaded manner from left to right when a join variable value retrieved from the L_1, dataset is used to set a range scan key [*begin*, *end*] by the *setRange* operator for R_1, join variable value retrieved from the R_1 dataset tuple is used to set a range scan key for the R_2 dataset, and so on through to the last dataset participating in the join. The use of the *multiple join* $\bowtie_M$ operator is more effective than the use of several sequentially executed operators $\bowtie (L, R)$. At that, the more sets are participating in a *multiple join*, the higher its effectiveness.

5 Heuristics Based Query Execution Plan Optimization

Our system employs a static query optimizer based on heuristics and performing the transformation of an initial query execution plan D_0 into an equivalent plan D_1. These heuristics do not use preliminary calculated database statistics.

"Leaf iterator constants shift" heuristic. A similar heuristic was discussed in [11]. The optimizer looks for constants in triple patterns and "shifts" them within

those patterns to the left, thus ensuring the use of sorted index structures. It is applicable only for the leaf iterators.

"Transform Cartesian product to natural join" heuristic. The optimizer tries to turn the *Cross-product* (*Cartesian product*, $\times$) [15] operator into one of our instantiations of the *join* ($\bowtie$) operator enabling the use of index scan (e.g., *zig-zag join*). Such transformation will be possible on the condition that tuples returned by at least either of the two arguments of the $\times$ operator can be sorted. The transformation pattern is: $\sigma_F(\times(L, R)) \Rightarrow \bowtie (L_{order_1}, R_{order_2})$, where the σ_F operator stands for the selection (*filter*) [15]. L and R are arguments of the $\times$ and $\bowtie$ operators that designate the joined multisets. The $order_i \in \{sorted, unsorted\}$ index under an argument signifies the $\bowtie$ operator argument sorting flag, *sorted* signifies a sorted dataset, and *unsorted* signifies an unsorted dataset. In order to apply $\bowtie$ with sorted arguments, the preliminary sorting of datasets must be ensured by any of the following methods: either by choosing an appropriate index, or by the multiset sorting operator τ_L [15].

"Reorder joins" heuristic. This heuristic works with the *natural join* operators. The transformation patterns are performed with the purpose of decreasing the size of unsorted argument datasets residing on preceding branches of the query execution plan tree by means of their relocation to the higher level. The transformation patterns in this case are: $\bowtie (\bowtie (L_{1unsort}, R_{1sort})_{sort}, R_{2sort}) \Rightarrow \bowtie (\bowtie (L_{1sort}, R_{2sort})_{unsort}, R_{1sort})$ and $\bowtie (L_{2sort}, \bowtie (L_{1unsort}, R_{1sort})_{sort}) \Rightarrow \bowtie (\bowtie (L_{1sort}, L_{2sort})_{unsort}, R_{1sort})$. Here and further indexes *unsort* and *sort* stand for unsorted and sorted multisets.

"Sort Minus arguments" heuristic. The transformation is aimed at the replacement of the complete scan of the right R and left L multiset entries of the *minus* [1] operator by the index search operations. The transformation patterns in this case are: $minus(L_{unsort}, R_{unsort}) \Rightarrow minus(L_{unsort}, R_{sort})$, and $minus(L_{unsort}, R_{unsort}) \Rightarrow minus(L_{sort}, R_{sort})$. This transformation is achieved by preliminary reordering variables, selecting proper indices, or sorting the right R and, ideally, left L multiset entries.

"Sort outer join arguments" heuristic. The transformation is aimed at the replacement of the complete lookup of the right R and left L relation-argument entries of the *left outer join* ($\overset{\circ}{\bowtie}_L$) [15] operator by the index search operations. The transformation patterns in this case is: $\overset{\circ}{\bowtie}_L (L_{unsort}, R_{unsort}) \Rightarrow \overset{\circ}{\bowtie}_L (L_{unsort}, R_{sort})$. This transformation is achieved by preliminary reordering variables, selecting proper indices, or sorting the right R and, ideally, left L relation-argument entries of the $\overset{\circ}{\bowtie}_L$ operator.

"Remove unrequired reordering" heuristic. The transformation is aimed at the removal from the query execution plan tree of a redundant iterator that performs the reordering of the columns of a resultant relation, should these resultant

relation columns be already arrayed in the required order. The transformation pattern is: $\tau_L(\Omega) \Rightarrow \Omega$. Here, τ_L is a sorting operator [15], L is a list of the relation Ω attributes, by which the sorting of this relation is performed.

"Convert multiple joins to one multiple join" heuristic. Given that there exists a combination of several nested *multiple join* operators (including *natural join*) with identical sorted join variables, they are transformed into a single *multiple join* operator. The transformation pattern in this case is: $\bowtie (\Omega_{1sort}, ..., \Omega_{nsort}) \Rightarrow \bowtie_M (M_{11sort}, ..., M_{1ksort}, ..., M_{n1sort}, ..., M_{nmsort})$. Here Ω_{isort} appears as $\bowtie (M_{i1sort}, ..., M_{ipsort})$, where M_{ij} is either a dataset or, in its turn, a nested operator join. All nested operators join are recursively replaced by a sequence of their arguments.

"Execute the simplest union first" heuristic. If a SPARQL expression consists of several *Union* sections, a query execution plan tree needs to be reordered to ensure that the sections containing simpler statements are executed first. For example, sections are considered "simpler" if they contain the *Limit* operator and do not contain the *Filter* operator.

"Move Projection closer to leafs" heuristic. The heuristic tries to place a multiset projection operator π [15] before sorting operator τ_L (*Order by*) and operator δ [15] performing the removal of duplicate tuples from that multiset. Such a technique allows to decrease the consumption of RAM used by the π and τ_L. The transformation patterns in this case are: $\pi_{j1,...,jn}(\delta(\Omega)) \Rightarrow \delta(\pi_{j1,...,jn}(\Omega))$, and $\pi_{j1,...,jn}(\tau_L(\Omega)) \Rightarrow \tau_L(\pi_{j1,...,jn}(\Omega))$.

"Sink Distinct down" heuristic. If a query execution plan graph includes the duplicate tuple removal operator δ_0, the heuristic evaluates the possibility of executing new δ_i operators missing in the initial execution plan closer to the leaf iterators for those variables that belong within tuples processed by the δ_0 operator, i.e., high level *Distinct* produce new *Distincts* on low levels. The transformation pattern is: $\delta_0(\Omega) \Rightarrow \delta_1(op_1(\delta_1(...(op_k(\delta_k(...)))))))$. Here, op_k stands for any acceptable operators of the query execution plan.

"Merge Distinct with Sorting" heuristic. If a *Select* clause contains solution modifiers *Distinct* and *Order by* [1], we substitute a new iterator performing simultaneously the duplicate tuple removal and sorting functions for the *Distinct* and *Order by* operators. This method allows to decrease the *Select* execution time, as well as to halve the demand for operative memory. The transformation pattern in this case is: $\tau_L(\delta(\Omega)) \Rightarrow \delta\tau_L(\Omega)$. Here, $\delta\tau_L$ stands for the operator simultaneously performing the duplicate tuple removal and sorting functions.

"Set sort buffer limit" heuristic. If the *Order by* sorting is present within a SPARQL expression, and the *Limit* limit is set there for a number of returned results, then a buffer is created, which is an orderly sorting queue with a size specified in the *Limit* for storing sorted resultant tuples. If, in accordance with

the sorting conditions, a new tuple read out by an iterator gets into a range of tuples already sitting within the buffer, it will be inserted into an appropriate position of the queue. When the sorting buffer becomes overflowed, surplus tuples are removed starting from the end of the queue. The sorting process finishes simultaneously with a query result completion, when calls to a temporary sorting table are entirely absent.

"Move filters closer to leafs" heuristic. If a query execution plan tree contains the *natural join*, *outer join*, *Cartesian product*, sort (*Order by*) operators, the heuristic tries to move filter in the query execution plan tree closer to the leaf nodes, placing it before the *natural join*, *outer join*, *Cartesian product*, sort operators. The transformation pattern in this case is: $\sigma_F(...op(L_1, R_1)...) \Rightarrow op(...(...\sigma_{F1}(\Omega_1)...\sigma_{F2}(\Omega_2)...)...)$, where the σ_F operator stands for the selection (*filter*), while the op operator stands for one of the $\bowtie$ (*natural join*), $\overset{\circ}{\bowtie}_L$ (*left outer join*), $\times$ (*Cartesian product*), τ_L (*Order by*) operators.

"Chose optimal distinct algorithm" heuristic. To implement the *Distinct* operator, we use two mechanisms, namely, the *hash map* table or *rb-tree* structure. If a query execution plan tree contains the multiset duplicate tuple removal operator δ (Distinct solution modifiers), an attempt at using the *hash map* structure instead of *rb-tree* is made to the extent possible. The *hash map* works faster than *rb-tree*, however, its employment in our implementation necessitates the *Vocabulary* identifiers for every variable, which is not mandatory in case of *rb-tree*. The use of *hash map* and dereference of variables are deemed possible when tuples do not contain variables obtained by means of the *Bind* operator [1].

"Merge join with filter" heuristic. The heuristic tries to replace two iterators of the query execution plan tree, the *join* iterator and nested into it *filter* iterator, by the unified iterator join with the simultaneous *filter* execution according to the filtering condition. Such a replacement improves the efficiency of the query execution. The transformation pattern, subject to the specified conditions, is: $\bowtie_M (\sigma_F(R_1), R_2, ..., R_n) \Rightarrow \bowtie_{M\sigma_F} (R_1, R_2, ..., R_n)$, where the σ_F operator stands for the selection (*filter*), while the $\bowtie_{M\sigma_F}$ operator stands for the *multiple join* operator combined with the R_1 set filtering.

"Replace join with multiple join" heuristic. The *multiple join* operator can be used instead of the join operator even in case of just two sets of data-arguments. Even with two arguments *multiple join(unsorted, sorted)* is faster than *join(unsorted, sorted)*, as *multiple join* uses the *lowerBound(key)* method to specify a dataset scan range instead of the *setRange(key)* method. (It was mentioned earlier that the *lowerBound* method works faster than *setRange*). The transformation pattern is $\bowtie_{replaced} (L_{unsort}, R_{sort}) \Rightarrow \bowtie_M (L_{unsort}, R_{sort})$.

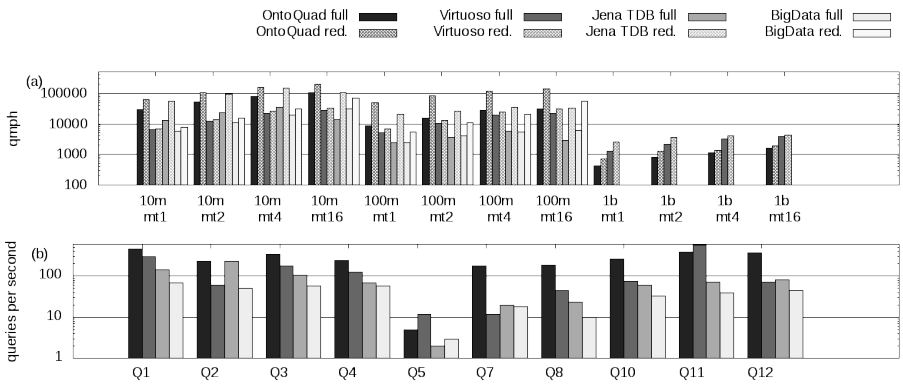

Fig. 3. (a) is total benchmark runtime in query mixes per hour (qmph) for the tested triple stores, dataset sizes and levels of parallelism on a logarithmic scale. (b) shows individual query runtime in queries per second for the 100m data set on a logarithmic scale..

6 Evaluation

In this section we present the evaluation of the OntoQuad RDF store. We benchmarked our system using the Berlin SPARQL Benchmark (BSBM) using the Explore use case of specification V3.1. The Explore use case of BSBM simulates an e-business use case, where a user explores a dataset about products and their features linked with other types, like reviews and offers. Such a browsing session is referred to as a query mix and consists of 25 SPARQL queries.

For comparison, we also tested Virtuoso 6.1.6, Jena TDB (Fuseki 0.2.7) and BigData (Release 1.2.2). The benchmark machine is powered by one quad-core Intel i7-3770 CPU with 32GB of RAM. The storage layer is comprised of 2x 2 TB 7200rpm SATA hard drives, configured as software RAID 1. All systems were configured to use 22GB of main memory.

In our benchmark efforts we varied the database size (10 million triples, 100 million triples and 1 billion triples), used different levels of concurrencies with 1, 4, 8, 16 parallel clients. Each store was warmed up with 2000 query mixes before the actual testing was performed. Also, we varied the set of queries used, as for large query sets BSBM query 5 dominates query run time due to its non-linear performance[4]. Therefore, one run with the full BSBM V3.1 query set and a reduced query set omitting query 5 was performed.

The overall benchmark results of these runs can be seen in Fig. 3 (a) where the total performance is given in query mixes per hour. In general, OntoQuad performs well in the comparison with the other state-of-the-art triple stores. In both the 10m and the 100m datasets OntoQuad is the best performing triple store, with performance gains up to an order of a magnitude. For the 1b dataset Virtuoso delivers the best performance, although the gap closes with increasing thread count. At all sizes we observe a near linear gain in performance for OntoQuad with increasing thread count, up to the point where the CPU is saturated.

[4] c.f.: http://lod2.eu/BlogPost/1584-big-data-rdf-store-benchmarking-experiences.html

In Fig. 3 (b) we display for the 100m dataset the individual query performance. Our efficient join execution strategy is reflected in the performance levels we achieve in most queries, but is especially well observer in query 7. As a counter-example, query 11 does not contain a join, and is one of the two queries where OntoQuad is outperformed. Query 5 is an example where there is still room for improvement. Here a large intermediate result has to be generated, which is currently not yet well supported. Virtuoso executes a more efficient strategy here.

7 Conclusions and Further Work

In this paper we presented the native RDF DBMS OntoQuad and its internal architectural principles which are based on the vector quadruples representation model. We described the efficient implementation by means of index structures as well as static SPARQL query optimization based on a set of heuristics. The presented approach ensures the achievement of query processing speeds that are comparable to the results of other state-of-the-art RDF database management systems. In our further elaboration on OntoQuad and continuation of the work, we intend to complete the following tasks: (a) database structure optimization at the data storage level by means of removing some redundant links between the quadruple data representation levels; (b) development of heuristics for query execution plan optimization in order to ensure the optimal BI performance of the BSBM test; (c) addition of a statistical estimator unit to the optimizer for evaluating performance efficiency of links and operators; (d) creation of the data store/DBMS cluster structure oriented towards the multi-platform processing of very-large-scale RDF data with regard to the development of Big Data-type semantic stores.

References

1. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Technical report, W3C Recommendation, 2013.
2. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In LA-WEB, 2005.
3. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In ISWG/ASWG, pages 211-224, 2007.
4. Harth, A., Decker, S.: Yet Another RDF Store: Perfect Index Structures for Storing Semantic Web Data With Context, DERI Technical Report, 2004.
5. Baolin, L., Bo, H.: HPRD: A High Performance RDF Database. In NPG, pages 364-374, 2007.
6. Weiss, C., Karras, P., Bernstein, A.: Sextuple Indexing for Semantic Web Data Management. PVLDB, 1(1):1008-1019, 2008.
7. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In VLDB, pages 411-422, 2007.

8. Wood, D., Gearon, P., Adams, T.: Kowari: A Platform for Semantic Web Storage and Analysis. In XTeGh, 2005.
9. Neumann, T., Weikum, G.: The RDF-3X Engine for Scalable Management of RDF Data. Journal: The Vldb Journal - VLDB , vol. 19, no. 1, pp. 91-113, 2010.
10. Neumann, T., Weikum, G.: RDF-3X: a RISC-style Engine for RDF. PVLDB, 1(1):647-659, 2008.
11. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. WWW 2008: 595-604
12. Hartig, O., Heese, R.: The SPARQL Query Graph Model for Query Optimization. ESWC 2007: 564-578.
13. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., eds.: Proceedings of the 5th International Semantic Web Conference. Volume 4273 of Lecture Notes in Computer Science, Springer (2006).
14. Gomathi, R., Sathya, C.: Efficient Optimization of Multiple SPARQL Queries. IOSR Journal of Computer Engineering (IOSR-JCE) e-ISSN: 2278-0661, p- ISSN: 2278-8727 Volume 8, Issue 6 (Jan. - Feb. 2013), PP 97-101, www.iosrjournals.org.
15. Garcia-Molina, H., Ullman, J.D., Widom J.: (2002). Database Systems: The Complete Book. Prentice Hall, Upper Saddle River, NJ.ISBN 0130319953.
16. Stonebraker, M., Çetintemel, U.: "One Size Fits All": An Idea Whose Time Has Come and Gone. In Proceedings of the International Conference on Data Engineering (IGDE), 2005
17. Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., Zdonik, S.: One Size Fits All? - Part 2: Benchmarking Results. In Proc. CIDR, 2007.
18. Antoshenkov, G., Ziauddin, M.: Query Processing and Optimization in Oracle Rdb. VLDB J. 5(4): 229-237 (1996).
19. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2): 73-170 (1993).