Open Access

Open Journal of Databases (OJDB)
Volume 3, Issue 1, 2016

http://www.ronpub.com/ojdb
ISSN 2199-3459

Runtime Adaptive Hybrid Query Engine based on FPGAs

Stefan Werner [A], Dennis Heinrich [A], Sven Groppe [A],
Christopher Blochwitz [B], Thilo Pionteck [C]

[A] Institute of Information Systems, Universität zu Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany,
{werner, heinrich, groppe}@ifis.uni-luebeck.de

[B] Institute of Computer Engineering, Universität zu Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany,
blochwitz@iti.uni-luebeck.de

[C] Institute for Information Technology and Communications, Otto von Guericke University Magdeburg,
Universitätsplatz 2, 39106 Magdeburg, Germany, thilo.pionteck@ovgu.de

Abstract

This paper presents the fully integrated hardware-accelerated query engine for large-scale datasets in the context of Semantic Web databases. As queries are typically unknown at design time, a static approach is not feasible and not flexible to cover a wide range of queries at system runtime. Therefore, we introduce a runtime reconfigurable accelerator based on a Field Programmable Gate Array (FPGA), which transparently incorporates with the freely available Semantic Web database LUPOSDATE. At system runtime, the proposed approach dynamically generates an optimized hardware accelerator in terms of an FPGA configuration for each individual query and transparently retrieves the query result to be displayed to the user. During hardware-accelerated execution the host supplies triple data to the FPGA and retrieves the results from the FPGA via PCIe interface. The benefits and limitations are evaluated on large-scale synthetic datasets with up to 260 million triples as well as the widely known Billion Triples Challenge.

Type of Paper and Keywords

Regular research paper: *Semantic Web, Query Processing, Query Engine, Hardware Accelerator, Field Programmable Gate Array, FPGA*

1 Introduction

Nowadays it is a fact that more and more (unstructured) data is stored and analyzed in several areas of research and industry, motivated by political and security reasons (surveillance, intelligence agencies), as well as economical (advertisement, social media) or medical matters [17]. In order to enable machines to automatically analyze (possibly not well or completely defined) data, the idea of the Semantic Web was created [4]. Therefore, metadata is used to describe and link any kind of data and resources but also allows the description of arbitrary complex concepts and models. Besides others, the *Linking Open Data* (LOD) project [37] aims to publish new and interconnect existing open datasets. As a result the LOD cloud consisted of more than 30 billion triples and more than 500 million links in 2011[1] (*Big Data*).

Besides suitable data structures, optimized hardware

[1] `http://lod-cloud.net/state/` (Version 0.3, 09/19/2011) - accessed: 05/11/2016

is necessary to store and process this vast amount of data. Whereas persistently storing of these massive data is hassle-free, the processing and analysis within a reasonable time frame becomes more and more difficult. In order to cope with these problems, in the last decades intensive work was done to optimize database software and data structures [20, 49, 22, 18]. Furthermore, technological advances enabled shrinking feature size to increase clock frequency and thus the overall performance.

However, nowadays these approaches are getting close to their limits (*power wall* [39]) and in the last years the trend evolved to multi/many-core systems in order to increase performance [5]. Additionally, these systems are not assembled with homogeneous cores, but rather are composed by heterogeneous and specialized cores which compute a specific task efficiently. The main issue of such systems is that these specialized cores cannot be used in applications showing a huge variety in processing. Widely available Field Programmable Gate Arrays (FPGA) with the capability of (partial) runtime-reconfiguration [44] are able to close the gap between the flexibility of general-purpose CPUs and the performance of specialized hardware-accelerators.

Although the presented methodologies are not limited to Semantic Web database systems, we develop a hybrid hardware-software system to enhance query performance on large-scale Semantic Web datasets. The proposed architecture allows the user to retrieve specific information from Semantic Web datasets by writing a query, which is automatically adapted and executed on a runtime-reconfigurable FPGA (see the basic architecture in Figure 1).

This paper is an extension of our previous work [54] but overcomes the reported performance issues by introducing virtual streams between host and FPGA to enable implicit synchronization. Instead of results obtained by simulation, this paper gives a comprehensive performance analysis on large-scale

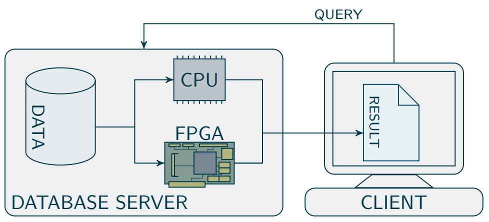

Figure 1: Basic architecture of the hybrid hardware-software system
(Explanation: The client application sends a query to the hybrid database server which transparently determines the result using an FPGA-based hardware accelerator.)

datasets. Additionally, the architectural overview provides more details about FPGAs and the collaboration between software and hardware in the hybrid system.

To the best of our knowledge this is the first hybrid hardware-software system which fully-automatically executes queries with an arbitrary number of operators[2] on large-scale Semantic Web datasets such as the Billion Triples Challenge [24] on an FPGA.

The remainder of this work is structured as followed: Section 2 outlines related work in the area of hardware-accelerators in database systems. The architectural overview and the general work flow of our proposed system is given in Section 3. The automated composition of the configuration suitable for the FPGA is described in Section 4. Section 5 evaluates the feasibility of the presented approach and Section 6 discusses open issues and future directions.

2 Related Work

In the following we review the recent work in the field of database tasks using modern hardware architectures. Already in the early days of databases the idea of a tailor-made database machine came up [36, 14, 13]. However, at this time the technological capabilities were limited and thus these approaches were not economically successful. Nowadays, since clock frequency growth reaches its limits and the availability of new hardware architectures, these and new ideas get a fresh impetus [5, 39].

2.1 Smart Storages

While CPU performance steadily increased for the last decades, the latency of main-memory access turned out to be a bottleneck known as the *memory wall* [38]. This imbalance exacerbates when it comes to persistent storages such as disk-based hard drives (HDDs) [35]. Although the latency is significantly reduced by flash-based *solid state drives* (SSDs), the disk bandwidth remains considerably low. With respect to databases, typically huge amounts of data are transferred to the CPU but often most of the data is irrelevant or is only a part of an aggregation. *Smart SSDs* provide an embedded processor, fast internal memory and multiple I/O channels resulting in a high performance on concurrent access patterns [33, 32]. Therefore, Do et al. [15] use the SSD's in-device processing capabilities to perform selection already in the SSD before transferring unnecessary data. Seshadri et al. [48] present a prototype which allows to implement application-specific data access utilizing the internal SSD processing resources.

[2]Only restricted by FPGA resources

2.2 General Purpose Computing on GPUs

Graphics processing units (GPUs) are specialized hardware accelerators to create and manipulate images stored in memory (frame buffer) to be shown on a display. Their internal structure is highly parallel, and this makes them interesting also for other fields than image processing. The methodology of performing general computations on GPUs is referred as *general purpose computing on GPUs* (GPGPU).

Sun et al. [50] show how to turn these characteristics into an hardware accelerator for spatial selections and joins. *GPUTeraSort* [19] utilizes a GPU as a co-processor for solving sorting tasks on huge datasets. He et al. provide extensive work on relational query co-processing using GPUs [26, 25, 27, 28, 10]. One major problem, which remains unsolved, is the transfer of data to the GPU. Fang et al. [16] reduce the overhead of data transfers by executing database compression techniques on GPUs. Another hybrid query processing engine using a GPU is shown by Breß [8, 7, 6].

2.3 Reconfigurable Computing

Reconfigurable architectures provide the post-fabrication programmability of software and the spatial parallelism of hardware [23]. Due to the availability of *Field-programmable Gate Arrays* (FPGAs), reconfigurable computing becomes more and more attractive and affordable. Typically, these devices obtain their performance advantages rather by inherent parallelism than high clock frequencies.

Mueller et al. [40, 41] propose the component library and compositional compiler *Glacier*. It takes a continuous query for data streams and generates a corresponding file written in the hardware description language VHDL. After the time-consuming translation of the VHDL description to an FPGA configuration, the query can be programmed on an FPGA in order to accelerate the evaluation on data streams. Consequently, this approach is only suitable for a known query set. Additionally, the library does not cover join operators.

Teubner et al. [51] present a window-based stream join, called Handshake join. The approach lets the items of two data streams flow by in opposite directions to find join partners with each item they encounter. All items in a predefined window are considered in parallel to compute an intermediate result. Due to the window-based architecture and since the window size is limited by the chip size, it cannot be guaranteed that the result contains all possible join partner of the two datasets.

IBM's Netezza FAST engines [30] uses FPGAs to reduce the amount of data to be transferred from (persistent) memory to the CPU by early execution of projection and restriction. Furthermore, uncompressing data at wire speed increases the read throughput and thus reduces the drawback of hard disks.

As the system does not support a modular composition of complete queries, Dennl et al. [11, 12] present concepts for on-the-fly hardware acceleration of SQL queries in the relational database MySQL. The authors focus on restriction and aggregation operators and thus cannot execute complete queries on the FPGA. However, in order to evaluate more complex queries (including joins) additional views are created to represent partial results and the proposed hardware-software system achieves promising speed-up gains.

Becher et al. [3] extend this approach to an embedded low-energy system-on-chip platform and add more complex operators (e.g., Merge Join and sorting of small datasets). For a simple query including one join they achieve a comparable performance but higher energy efficiency than a standard x86-based system. Additionally, they provide a theoretical model for their operators to estimate the performance.

Casper et al. [9] explore accelerating in-memory database operations with focus on throughput and utilization of memory bandwidth during sorting. The presented system performs an equi-join after sorting of two tables.

Woods et al. [58, 57] present *Ibex*, an intelligent storage engine for the relational database MySQL that supports off-loading of complex query operators using an FPGA. Comparable to Netezza FAST engines but more flexible, the FPGA is integrated into the data path between data source and host system. István [31] beneficially uses this system to generate statistics and histograms as a side effect of data movement in the data path. Typically, the generation of histograms is computing intensive, but they are important for query planning.

Sadoghi [45] and Najafi [42, 43] describe a reconfigurable event stream processor based on an FPGA. It supports selection, projection and window-joins on data streams.

Heinrich et al. [29] propose a hybrid index structure, which stores the higher levels of a B$^+$-tree including the root on an FPGA. The lower levels including the leaves are located on the host system. As a result, the access on frequently entered higher levels is accelerated. The FPGA returns an entry point (address) from where the software system continues the search.

In the context of Semantic Web databases we develop a hybrid hardware-software system, which transparently transforms and executes SPARQL queries on a run-time reconfigurable FPGA. In previous works we have presented our approaches to implement and execute the join operator [53, 56] and filter expressions [55] on an

FPGA. Whereas these operators have been evaluated in isolation, this work presents the fully automated and transparent composition and execution of complete operator graphs on an FPGA. As a result the user is able to simply write an arbitrary SPARQL query in the GUI of LUPOSDATE [21], which is automatically transformed into a configuration suitable for the FPGA and evaluated on it. Finally, the result is displayed by the host system to the user.

3 Architectural Overview

In the following we introduce the basic internals about Field Programmable Gate Arrays (FPGA) and some fundamentals of Semantic Web databases. Furthermore, we present the overall architecture of the hybrid hardware-software system and give an insight into how our proposed system synchronizes between host and FPGA during query evaluation.

3.1 Field Programmable Gate Array (FPGA)

In the following the Xilinx Virtex-6 [60] FPGA is described as it is used in this work. The overall structure of different FPGAs is conceptually the same and differs typically in structure size, amount of provided logic resources and dedicated hardware cores (such as digital signal processors (DSP) or general-purpose CPUs). The FPGA die is organized in Configurable Logic Blocks (CLB [61]), which each consists of 2 slices. Each slice contains 4 Lookup Tables (LUTs), 3 multiplexer, a carry chain and 8 flip-flops (FF). Each LUT can be used as one 6-input with one output but can also be used as two 5-input LUTs (using common logic inputs) with two separate outputs. The output of the LUTs can be stored in the FFs.

Besides implementing combinatorial functions (*SLICEL*), the LUTs of between 25 to 50% of all slices can be used as distributed 64-bit RAM or 32-bit shift registers (*SLICEM*). Each slice is connected to a switch matrix which in turn is linked to several switch matrices nearby. Additionally, fast carry chains allow vertical data propagation from one slice to another above. This enables to combine slice to achieve complex functions. Figure 2 shows a very small segment of the total[3] CLB array. Within the array, columns of Block RAM (BRAM) and DSP blocks are located as well. Depending on the device, multiple BRAM blocks are placed in columns on the whole FPGA die. In case of the Virtex-6 (and others) each block has 36 Kb storage area but can be segmented into 2 independent 18 Kb BRAMs. Furthermore, Virtex-6 family supports various

[3]Typically hundreds of thousand

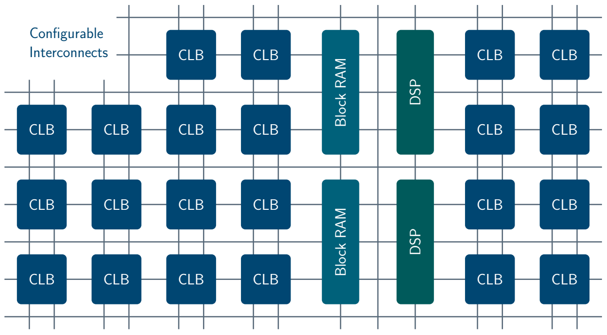

Figure 2: FPGA internals schematic - array of Configurable Logic Blocks (CLB)

memory-related interfaces such as DDR2 SDRAM, DDR3 SDRAM, RLDRAM II and QDRII+SRAM. Thus, FPGA architectures provide a flexible memory hierarchy with very small but very fast storage up to comparably bigger but slower memory with different possible granularities. In order to get a suitable configuration (called *bitfile*) for the FPGA, the intended hardware can be expressed in a hardware description language such as VHDL[4]. Besides replacing the whole design on the FPGA, Dynamic Partial Reconfiguration (DPR [62]) allows to replace some parts of the design at runtime. In this work, DPR is extensively used to exchange queries in this approach.

3.2 LUPOSDATE

LUPOSDATE [21] is an open source Semantic Web database system which allows the easy integration of Semantic Web technologies in any other application or the extension and contribution to LUPOSDATE itself. Among numerous other features, it provides several query engines, all (except the streaming engine) supporting full SPARQL 1.0 and SPARQL 1.1 [52]. The developed hybrid architecture in this work is based on and transparently incorporates with the LUPOSDATE system by introducing an FPGA-based query engine. Therefore, the next section introduces the basic data and query formats.

3.3 Data Representation and Queries

The *Resource Description Framework* (RDF [59]) is used as the basic data format in the Semantic Web to describe statements about web resources. The data is represented as RDF triples. Listing 1 shows an example RDF dataset consisting of two prefix definitions and two triples with a common subject. In general, each triple

[4]**VHDL**: **V**HSIC **H**ardware **D**escription **L**anguage (**VHSIC**: **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit)

Listing 1: RDF example (modified from SP^{2}B dataset [47])

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
<http://localhost/journals/Journal1/1940> dc:title        "Journal 1 (1940)"^^xsd:string  .
<http://localhost/journals/Journal1/1940> dcterms:issued "1940"^^xsd:integer               .
```

Listing 2: SPARQL query on the SP^{2}B dataset [47]

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?title ?yr
WHERE {
  ?doc dc:title ?title .
  ?doc dcterms:issued ?yr
}
```

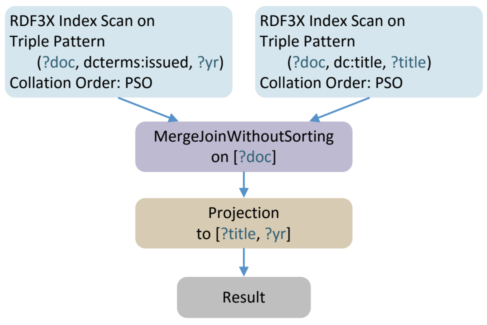
RDF3X Index Scan on Triple Pattern (?doc, dcterms:issued, ?yr) Collation Order: PSO
RDF3X Index Scan on Triple Pattern (?doc, dc:title, ?title) Collation Order: PSO
MergeJoinWithoutSorting on [?doc]
Projection to [?title, ?yr]
Result

Figure 3: Operator graph of the example query in Listing 2

?title	*?yr*	*?doc*

Figure 4: Bindings array of the example query in Listing 2

Table 1: Dictionary regarding literals in Listing 1 (without prefix substitution)

String	ID
<http:// localhost / journals / Journal1/1940>	0
dc: title	1
dcterms: issued	2
"Journal 1 (1940)"^^xsd: string	3
"1940"^^xsd: integer	4

Listing 3: Representing RDF triples of Listing 1 using IDs from Figure 1

```
0 1 3 .
0 2 4 .
```

consists of components *subject*, *predicate* and *object*, and is formally defined as $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, with the pairwise disjoint infinite sets I, B and L, where I consists of IRIs, B is the set of blank nodes and L contains literals.

A set of RDF triples is used as the data basis for SPARQL [52] queries. Listing 2 shows an example for a SPARQL query to retrieve all document titles and their year of publication in the SP^{2}B dataset [47]. The SELECT clause defines a projection list of variables to appear in the final result (i.e., the bindings of the variables *?title* and *?yr*). The WHERE clause contains two triple patterns. Matching triples lead to bindings of the variables. As the variable *?doc* appears in both triple patterns, both intermediate results will be joined (in this case with a Merge Join). Like in relational databases, the components of SPARQL queries are broken down to a set of nestable basic operators and are combined to an operator graph representing the given query. The resulting operator graph of the example query is shown in Figure 3.

Depending on the constant values in a triple pattern LUPOSDATE chooses a collation order for each index scan operator. As in both triple patterns the predicate is a constant value the collation order PSO is chosen. This means the data is primarily sorted by the predicate, secondarily by subject and tertiary by object. In fact, LUPOSDATE uses six indices, each for one of the six possible collation orders (SPO, SOP, etc.) of RDF triples, and thus is able to retrieve sorted triples efficiently for a given triple pattern. Intermediate results (bound values for the variables) are stored in *bindings arrays*. Each variable has a dedicated position in this array where its bound value is stored (see Figure 4).

Furthermore, the LUPOSDATE system uses a dictionary to map RDF terms into integer IDs [22] and thus each binding in the bindings array is an integer ID which refers to the actual string representation (see Figure 1 and Listing 3). Due to lower space consumption of the evaluation indices and a significantly smaller memory footprint of intermediate results during query execution, this feature is intensively used by LUPOSDATE. With respect to the FPGA design this is quite handy as handling arrays of integer values is rather easier than dealing with strings of variable length.

Table 2: Brief overview of relational operators (with input relations R, S)

Operator	Notation	Description
Filter (Selection)	$\sigma_c(R)$	Returns only tuples of R which fulfill formula c
Projection	$\pi_{v_1,\ldots,v_n}(R)$	Keeps only the variables $v_1, \ldots, v_n$ of the tuples of R
Cross Product	$R \times S$	Concatenates each tuple of R with each tuple of S
Equi-Join	$R \bowtie_{v_1,\ldots,v_n} S$	Concatenates all tuples of R with all tuples of S which are equal in the common join variables $v_1, \ldots, v_n$
Union	$R \cup S$	Returns all tuples of R and S
Limit	$limit_n(R)$	Returns first n tuples of R
Offset	$offset_n(R)$	Returns all except the first n tuples of R
Distinct	$\delta(R)$	R without duplicates
Order By	$\tau_{v_1,\ldots,v_n}(R)$	R sorted with respect to variables $v_1, \ldots, v_n$
Triple Pattern Scan	(i_1, i_2, i_3)	Returns triples satisfying the pattern

Besides the basic structure, consisting of SELECT/WHERE clause and triple pattern, other features such as FILTER, UNION and OPTIONAL are provided by SPARQL. Each SPARQL operator (except the index scan operator) can be expressed as relational algebra. Table 2 gives a brief overview of the important operators of the relational algebra.

3.4 Hybrid Work Flow

Figure 5 shows the general processing flow. First the user submits a query which is transformed into an operator graph by the LUPOSDATE system. On the operator graph several logical and physical optimizations are applied [22]. Afterwards the optimized operator graph is analyzed for unsupported operators[5]. In case such an operator was found the query is evaluated completely by the software engine on the host system. Besides implementing more operators on the FPGA, in the future we plan to break down this strict separation as well. This means that the FPGA processes as much as possible of the operator graph and the host system covers the remaining operators. If all operators are supported then our new extension traverses the operator graph and generates a VHDL file which represents the given query. A detailed description of this process can be found in Section 4. The full FPGA tool chain (synthesis, mapping, place & route) is applied on the VHDL description of the circuit. The resulting bitfile is programmed on the FPGA and the query is ready to be executed. Again, in case of any error (e.g., translation failed caused by lack of FPGA resources) the query is evaluated in software. This always preserves a running system. If no error occurred the query is executed on the FPGA. In fact, during execution the host and FPGA work in parallel. The host covers the following two tasks during query execution:

[5]e.g., sorting operators are not implemented yet

1. **Provision of input triples:** Depending on the triple patterns given in the query, a collation order for each index scan is chosen. The underlying data structure (B$^+$-Tree) iteratively returns all ID triples satisfying the triple pattern. The ID triples are written in a buffer on the host and block-wisely passed on to the PCIe engine.

2. **Retrieval and post processing of results:** Concurrently the host is requesting resulting bindings arrays by handing over an empty buffer to the PCIe engine. Usually the buffer contains multiple results. In further post processing steps the result's low level representation (array of integer) is packed into higher data structures (literal and bindings objects) in order to return the result back to typical software flow and its modules for presenting the result to the user or submitting it to another application.

3.5 Hybrid Architecture

Figure 6 shows the architecture of the hybrid system consisting of two parts: the host, which provides higher functions (such as user interface, query optimization, maintaining of evaluation indices, etc.), and the FPGA as an accelerator for query execution. The communication between both units is based on PCIe.

The FPGA is divided into two partitions: *static* and *dynamic*. The static partition contains modules which are independent of the actual query structure. Typically, those are modules for communication and memory interfaces. In this case it contains the PCIe Endpoint (EP) and a managing module, the Query Coordinator (QC). The main task of the QC covers delivery of incoming triples to the corresponding index scan operators in the dynamic partition as well as retrieval and serialization of bindings arrays (forming the final result). The dynamic partition can be reconfigured

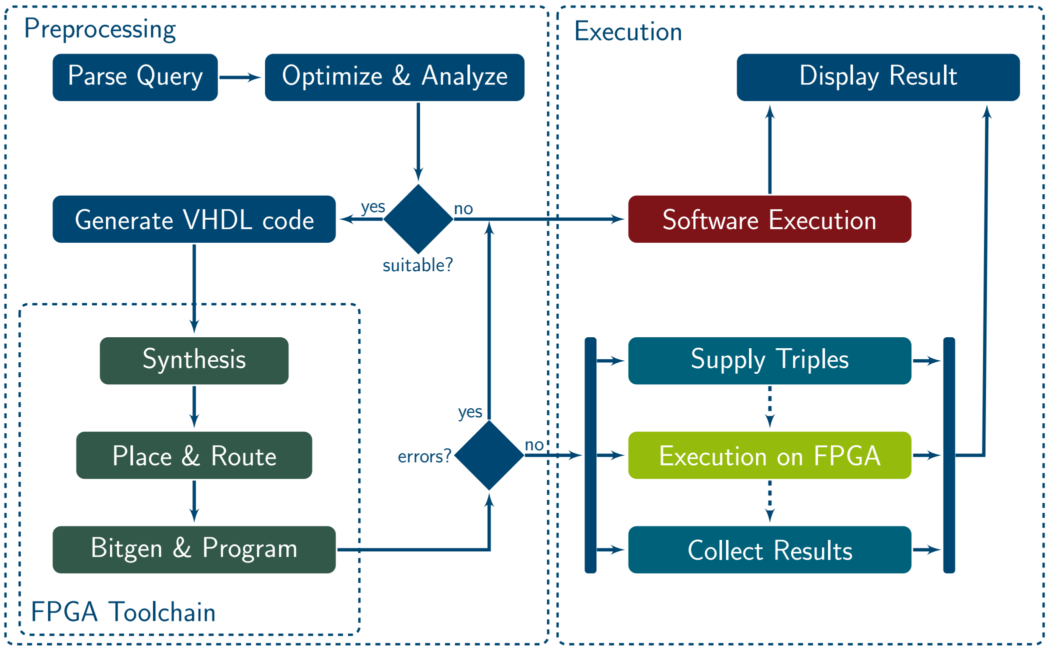

Figure 5: Flow chart of the hybrid system

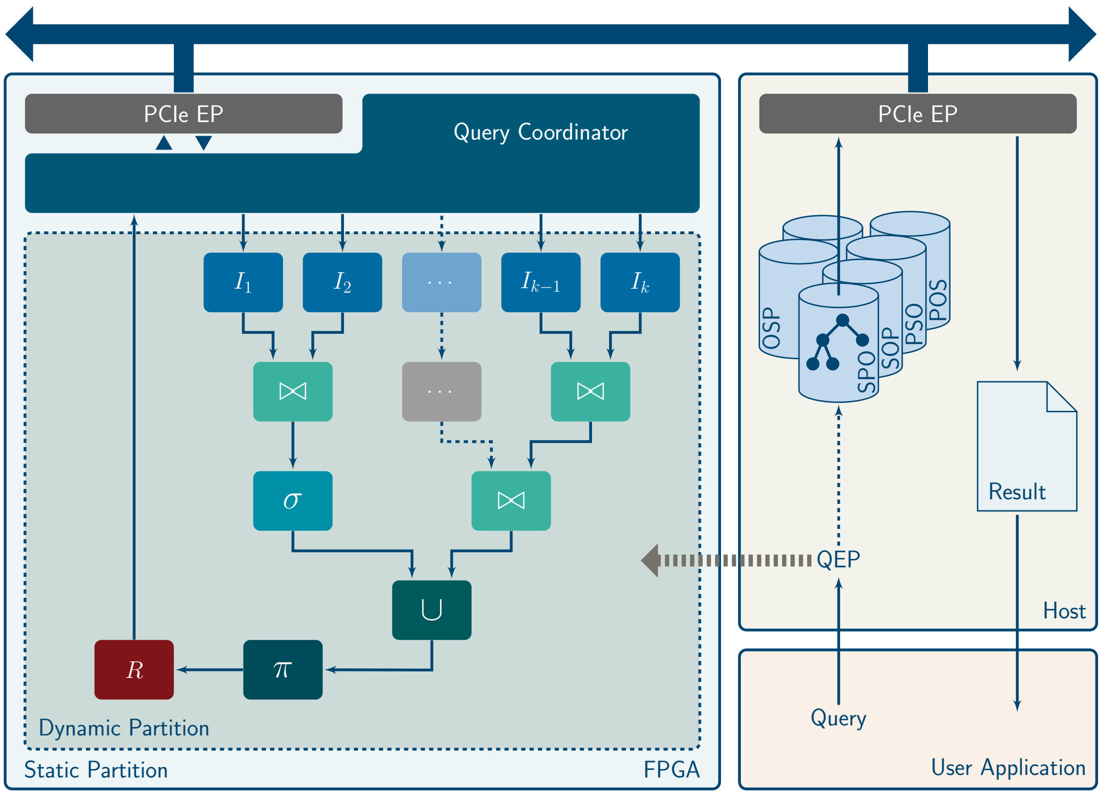

Figure 6: Hybrid Query Execution Engine (EP=Endpoint, QEP=Query Execution Plan)

Table 3: Parts of the VHDL template for the dynamic partition

Static signals & constants definition
Dynamic signals & constants definition
Dynamic signal assignment
Dynamic mapping of index scans
Static instantiations of entities
Dynamic instantiations of entities
Static *glue* logic

at runtime with a bitfile representing any arbitrary operator graph (limited by chip space).

All operator graphs contain index scan operators I_n with $n \in \{1..k\}$ and k fixed, but adjustable at system setup time (currently $k = 8$). Each index scan represents a triple pattern (with possibly bound components) and maps the incoming triple components into the corresponding variable positions in the bindings array. Succeeding operators (such as Join, Filter, Union, etc.) consolidate and filter partly bounded bindings arrays to combined final results. These results are returned to the QC in the static partition in order to transfer them to the host. On host-side, the result is materialized in higher data structure for further processing such as displaying the result to the user or delivery to the calling application.

Contrary to our previous approach, there is no additional explicit control flow between host application and hardware unit necessary. This is achieved by using virtual streams which logically divide the PCIe interface. In the FPGA and on the host, each stream provides a dedicated interface including dedicated buffer (BRAM respectively main memory). For incoming triple data, each stream is associated with one index scan operator. The result is always sent back on the first stream. Remaining streams can be used in the future to support processing of multiple queries on the FPGA at the same time. In the following section we describe how to automatically obtain a configuration for the dynamic partition.

4 Automated Composition

The configuration of the dynamic partition is generated automatically for each query. We developed a VHDL template and a pool of operators to compose the VHDL file which represents the query. In turn the template consists of a static and dynamic part as well (shown in Table 3). In the following, a detailed description for each part of the template is given.

4.1 Template - Static Parts

Each query contains a result operator which is connected using the signal *results* of the record type `op_connection` (see Listing 4). It is used as an interface between the dynamically generated operator graph and the static code which takes the query results and serializes them into the PCIe TX engine. The record type `op_connection` plays an important role in the dynamic part as well and will be explained in detail in Section 4.2. Additionally, the static instantiation of entities covers a reset generator, a cycle counter (for debugging purposes and to evaluate the raw FPGA performance) and the Query Coordinator (QC). The QC is the interface between the PCIe Endpoint in the static partition and an adjustable[6] number of index scans.

Listing 4: Connection record

```
type op_connection is record
  read_data  :  std_logic;
  data       :  std_logic_vector(DW-1 downto 0);
  valid      :  std_logic;
  finished   :  std_logic;
end record op_connection;
```

4.2 Template - Dynamic Parts

The template is analyzed for the dynamic parts (see Table 3) by LUPOSDATE at startup. Basically the dynamic parts are marked with predefined markups as comments. LUPOSDATE searches for those markups and replaces them with VHDL code as follows.

4.2.1 Operator Instantiation and Interconnects

The operator template is shown in Figure 7. It defines the input and output signals which need to be implemented by each operator. Operators can have up to two preceding operators. If an operator needs only one preceding operator (e.g., filter) then only the left input is used. The second input is simply not used by the operator and not connected to any other operator by the automated composition algorithm. The following synthesis implicitly detects those unused signals and removes them accordingly. Furthermore, each operator has exactly one succeeding operator. The signals are grouped in such a way that the output of each operator can be used as an input for any other operator. Each group consists of (i) a vector *data* which corresponds to the bindings array, (ii) a *valid* flag which indicates the validity of *data*, (iii) a *finished* flag which indicates the

[6]We support up to 16 index scan operators

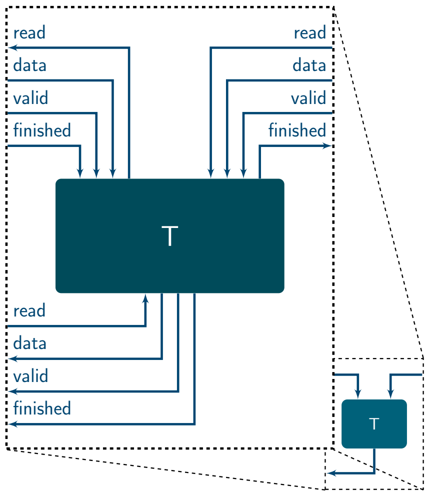

Figure 7: Common operator interface

end of *data*, and (iv) a backward flag *read* which notifies the proceeding operator that *data* was read. Reading the *data* implicitly invalidates the *data* until new data is provided by the preceding operator. It can be seen that one group corresponds to the previously defined record type `op_connection` (Listing 4). The data width (DW) of the signal *data* depends on the number of variables in the bindings array and is dynamically set for each query.

Thus, while traversing the optimized operator graph for each visited operator, an ID *X* is assigned and the signals *opXinput1*, *opXinput2* and *opXoutput1* are defined (see lines 1 to 3 in Listing 5). Additionally, an entity of a specific operator with type *OperatorType* is instantiated (line 5). In the port map (lines 11 to 27) the internal operator signals are connected with the previously defined `op_connection` signals. The generic map (lines 6 to 10) is used to parameterize this particular operator. All operators have the generics for the data and value width in common which correspond to the width of the bindings array and variables. Furthermore, each operator can have additional individual generics. These extensions will be described in Section 4.3. The actual wiring of two consecutive operators is outlined in Listing 6. As the output of operator *X* is used as (left) input of the operator *Y* this implies that *X* is the predecessor of *Y* (see lines 2 to 4). The *read_data* flag indicates operator *X* that operator *Y* has read the provided data and thus can provide the next data (see line 1).

4.2.2 Input Mapping

Contrary to our previous work [54] the input mapping of triples to index scan operator is realized implicitly

Listing 5: Operator instatiation

```
 1  signal opXinput1  : op_connection;
 2  signal opXinput2  : op_connection;
 3  signal opXoutput1 : op_connection;
 4  [...]
 5  operatorX : entity work.OperatorType(arch)
 6  generic map(
 7    DATA_WIDTH  => BINDINGS_ARRAY_WIDTH,
 8    VALUE_WIDTH => BINDINGS_WIDTH,
 9    --[... more operator specific generics ...]
10  )
11  port map(
12    [...]
13    left_read      => opXinput1.read_data,
14    left_data      => opXinput1.data,
15    left_valid     => opXinput1.valid,
16    left_finished  => opXinput1.finished,
17
18    right_read     => opXinput2.read_data,
19    right_data     => opXinput2.data,
20    right_valid    => opXinput2.valid,
21    right_finished => opXinput2.finished,
22
23    result_read    => opXoutput1.read_data,
24    result_data    => opXoutput1.data,
25    result_valid   => opXoutput1.valid,
26    finished_out   => opXoutput1.finished
27  );
```

Listing 6: Linking of two operators X and Y

```
1  opXoutput1.read_data <= opYinput1.read_data;
2  opYinput1.data       <= opXoutput1.data;
3  opYinput1.valid      <= opXoutput1.valid;
4  opYinput1.finished   <= opXoutput1.finished;
```

by assigning to each index scan one virtual stream. This suits better the streaming fashion of the whole architecture. The synchronization between the PCIe endpoints at host and FPGA is implicit as well. This is a great advantage if the triples of different index scans are not consumed at the same speed, and thus the buffer of the slower index scan gets filled on the FPGA side and more triples for this particular index scan must not be sent anymore until the buffer depletes. Assuming that all triples for all index scans are sent using the *same* stream this might cause that the triples of the blocked index scan are blocking the commonly used stream.

As a consequence, the other index scans receive no more triples and thus the whole query execution is blocked. Of course this can be avoided by an explicit and strict synchronization between host and FPGA, but causes additional delays and protocol overhead. Thus, we divide one physical stream into multiple virtual streams, each implicitly synchronized and not effecting other virtual streams. However, as all virtual streams share one physical interface consequently the bandwidth is shared too. The available bandwidth for each virtual stream is not preassigned which means if only one virtual stream is used to send data then this stream can utilize the full bandwidth.

4.3 Parametrization of Operators

As mentioned before each operator type can have individual generics. In the following the scheme to parametrize operators is outlined.

4.3.1 RDF3XIndexScan

The RDF3XIndexScan is the link between the QC and the inner operators. Typically the RDF3XIndexScan provides data triples s, p, o but a bindings array can have less or more than three variables and also not all of the three triple components might be necessary to evaluate the query. Thus, the main objective of this operator is to receive triples from the QC and map their required components to a position in the bindings array. Therefore the RDF3XIndexScan has three additional generic one-hot-coded bit vectors (`SUBJECT_POSITION`, `PREDICATE_POSITION`, `OBJECT_POSITION`). Each vector consists of as much bits as there are variables in the bindings array. During synthesis these vectors are evaluated following a simple scheme: If bit x is set in the bit vector of one triple component then this triple component is connected to position x in the bindings array. Unbound variables are initialized with an invalid value (0xFFFFFFFF).

4.3.2 Join

This operator joins the intermediate results of two preceding operators depending on one or more common join attributes. Similar to the RDF3XIndexScan, the position of the join attribute is determined by the one-hot-coded bit vector `JOIN_VECTOR`. As the structure of the bindings array is globally the same in the whole operator graph, only one set bit is necessary. However, it is possible that a join on more than one common variable is executed. Although this is not yet supported by the proposed system it is possible to simply add additional bit vectors for secondary, tertiary, etc. orders. In fact, there are several different algorithms for join computation such as Merge or Hash Join [53] but all are equipped with the same generics.

4.3.3 Filter

In previous work [55] we presented two approaches to implement the filter operator for Semantic Web databases. Taking the optimizer of LUPOSDATE into account we are able to break down complex filter expressions into multiple simple filter operators of the scheme *VALUE COND VALUE*, with *COND* as the condition (e.g., equality) and *VALUE* either a constant or variable. Specifically, this means that conjunctions of filter expressions result in a chain of simple filter operators each checking only one relational condition. In case of disjunction the operator duplication takes place and thus multiple disjunctive conditions are evaluated by simple filter operators in concurrent branches of the operator graph. In turn the intermediate results of two (or more) branches simply need to be unified in a lower level of the operator graph. As a result each filter operator is equipped with the following generics.

The generic `FILTER_OP_TYPE` describes the relational operation to be evaluated by the filter. Due to the mapping from strings to integer IDs our approach supports only *equal* and *unequal* comparators at the moment. However, if the dictionary (ID$\rightarrow$string) would be available on the FPGA also other conditions such as *greater/smaller than* are possible. Furthermore, we have to distinguish between expressions comparing a variable with a constant value and comparing two variables of the bindings array. Therefore `FILTER_LEFT_IS_CONST` is set if the left value is a constant. If so then the constant value is passed through the generic `FILTER_LEFT_CONST_VALUE` by setting the actual value to be compared. Contrary if the left value is not a constant then the one-hot-coded bit vector `FILTER_LEFT_VAR_POS` is considered. Like in previously described operators a set bit in this vector corresponds to the position of the variable in the bindings array. By simply replacing the term `LEFT` with `RIGHT` in the generics that scheme is applied for the right value of the filter expression as well.

4.3.4 Projection

The Projection carries out the SELECT clause of the SPARQL query. Therefore it is equipped with the bit vector `PROJECTION_VECTOR`. It has as much bits as the bindings array has variables. If bit x is set to '1' in the bit vector then the corresponding variable at position x in the bindings array remains in the result. Otherwise the corresponding variable is projected out.

4.3.5 (Merge) Union

The Union operator builds the union of the results provided by its two predecessors. It is not necessary that the same variables are bound at this point in the query execution. An extension, the Merge Union, requires sorted inputs and unifies the two input such that the result is still sorted. Therefore, it is equipped with the one-hot-coded bit vector `UNION_VECTOR`, to indicate regarding which variable the order has to be preserved. Similar to the generics of the join additional bit vectors might be added to enable secondary, tertiary, etc. orders. The simple Union has no additional generics.

4.3.6 Limit and Offset

The Limit operator is typically located directly before the result operator and forwards a specific amount of resulting bindings arrays. After its limit is reached (by simply counting) or its preceding operator indicates finish, it rises its finished flag which propagates through the result operator to the QC. As a consequence the QC will close the result stream to the host which can be detected on application level. In turn the Offset operator skips a specific number of the first resulting bindings arrays and simply passes the remaining bindings array to its successor. Combining Limit and Offset selects different subsets of the query result. Therefore, both operators are equipped with an integer generic, `LIMIT` respectively `OFFSET`, to set its corresponding value.

4.3.7 Unsupported Operators

At this stage, we support a subset of SPARQL 1.0 using the previously described operators. The Sorting and Distinct operators are not implemented so far. Both must temporarily store the whole intermediate result of their predecessors, and thus have enormous memory requirements which can not be satisfied by using only BRAM. Koch et al. [34] utilize the entire FPGA in their sorting architecture and thus is not applicable in our approach. However, extending our approach with additional memory interface such as DDR3 and with support for mass storage devices like SSDs, we will be able to implement these operators in the future.

The OPTIONAL operator (*left outer join*) can be derived from already implemented join operators. Furthermore, SPARQL tests such as *isIRI* and aggregation functions are not implemented, yet. Some redundant features like the SPARQL 1.1 paths (restricted to those without repetitions) can be partly covered by query rewriting. However, if an unsupported operator is detected during query optimization the proposed system always falls back to the software-only execution covering full SPARQL 1.1. Additionally, as a next step the operator graph could be partly located on the FPGA and on the host system. Latter executes the remaining not implemented operators. Updates are always performed by the host system.

5 Evaluation

This section describes the evaluation setup and analyzes the proposed architecture with respect to the query execution time and the resulting speedup compared to the software-only system.

5.1 Preliminaries and System Setup

The host system is a Dell Precision T3610 (Intel Xeon E5-1607 v2 3.0 GHz, 40 GB DDR3-RAM) which is equipped with a Xilinx Virtex-6 FPGA (XC6VHX380T) [60] board via PCIe 2.0 with 8 lanes. In previous work [54], we presented only results obtained by simulations on relatively small[7] datasets due to instabilities and performance lacks in our PCIe implementation. It is expected that the throughput significantly impacts the overall performance of the acceleration but still it was shown that even at lower throughput the presented approach outperforms the classical CPU-based execution. As described in the previous sections the overall architecture was re-factored in order to avoid delaying synchronizations between host and FPGA, and thus suits better to the streaming fashion of the query execution.

However, these changes do not allow direct comparisons to be made. The PCIe implementation is realized using the freely available Xillybus core [63]. The used FPGA board is equipped with an 8-lane gen2 PCIe interface. The theoretical possible bandwidth of this interface is noted with 4 GB/s. However, for the used Xilinx Virtex-6 a data rate of 400 MByte/s using only 4 PCIe lanes is reported[8]. Furthermore, the developers of Xillybus report a reduction to 200 MByte/s (due to overhead of the data link layer and TLP packet headers) and additionally mention that often processing the data on application level turns out to be the real bottleneck[9].

Therefore, we evaluated the Xillybus core in our environment with different buffer sizes at application level. Writing from the application to the FPGA we achieve throughputs between 100 and 300 MByte/s. Mostly it depends on the total amount of data. With a high amount of data the internal buffers are utilized intensively and thus the data transactions are more efficient. If a buffer is not filled then the actual data transfer will be initiated after a timeout[10] which results in a lower total throughput due to higher overhead compared to raw data. On the other hand, reading from the FPGA is significantly slower in a range of only 10 up

[7] 1,000 up to 1 million triples [54]

[8] `http://xillybus.com/pcie-download`

[9] `http://xillybus.com/doc/xillybus-bandwidth`

[10] Currently 5 ms; adjustable during core generation

to 200 MByte/s depending on the chosen buffer size on application level (10 to 10,000 32-bit values). However, we will show in the next sections that the PCIe interface, although not even closely utilized at its specification, is not the bottleneck of our architecture.

5.2 SP^{2}Bench SPARQL Performance Benchmark

As a first step in order to evaluate the presented approach systematically we use the SP^{2}Bench [46]. Besides example queries, it provides a data generator which is able to generate datasets with different triple cardinalities. The generated data itself is motivated by the project *Digital Bibliography & Library* (DBLP) and thus is supposed to mirror key characteristics of real world data.

For the following runtime analysis we use datasets with varying cardinalities starting at 1 million up to 262 million triple. Due to missing operators (e.g., sorting and distinct) we choose five SPARQL queries (inspired by the SP^{2}B queries) to show the feasibility of our approach. Query 1 consists of one join and a simple filter expression. Query 2 consists of two joins which can be executed independently. Both intermediate results are unified. Query 3 consists of three joins. Two of them can be executed independently as well, while the third join combines the intermediate results of the previous two operators. The last join operates in a pipelined fashion concurrently to the other two joins. Query 4 introduces an additional variable and thus another join, but the size of the result set is the same as Query 3. Query 5 is a further extension with one variable/join more and a smaller result size. The complete test queries can be found in the appendix. The size of the result depending on the input dataset size is shown in Figure 8 for each test query.

Table 4 gives an overview of the metrics used in the following performance evaluation. The execution time of the standard LUPOSDATE software system $Software$ is typically shown with the red line and each circle represents the average of 1,000 single executions with warm caches[11]. The execution times of the hybrid system are labeled with $FPGA$ but covers the whole processing time on host and FPGA *including* communication cost between them. Further, we differentiate between $FPGA_{full}$ and $FPGA_{post}$. $FPGA_{full}$ includes the time of the whole evaluation on the hybrid system and a final iteration through the obtained result. $FPGA_{post}$ includes the previously described post processing steps on the host (see Section 3.4). Setting each of the both execution times set in relation to the software-only approach results in the achieved speedup $Speedup_{full}$ respectively $Speedup_{post}$.

[11] Achieved by a single execution just before the actual performance evaluation

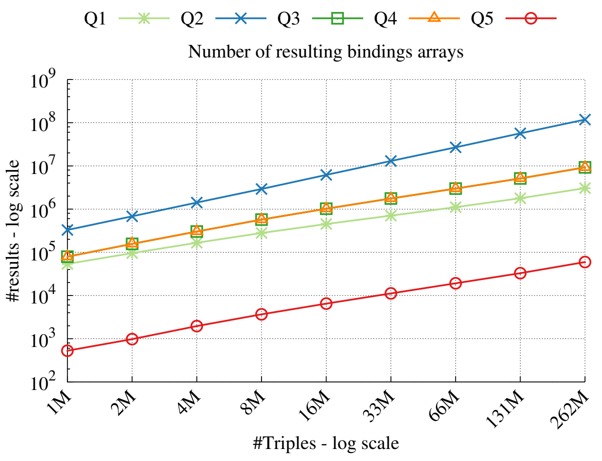

Figure 8: Result size of test queries according to different size of the SP^{2}B dataset [47]

Table 4: Performance metrics used in the evaluation of the software and hybrid system

Label	**Description**
$Software$	Query execution time using the standard LUPOSDATE software system. No FPGA is used.
$FPGA_{full}$	Query execution time of hybrid system including full iteration through result on host.
$Speedup_{full}$	$Software$ / $FPGA_{full}$
$FPGA_{post}$	Query execution time of hybrid system including post processing of results on host.
$Speedup_{post}$	$Software$ / $FPGA_{post}$

Figure 9 to Figure 13 show the execution times of the test queries for different dataset sizes. Regarding the very simple query 1 (Figure 9), the software-only and the hybrid approach scale linearly to the dataset. However, $FPGA_{full}$ grows slower on the hybrid system which results in an increasing speedup of up to 21X. However, post processing the result on the host causes an higher overhead with increasing result size shrinking the achieved speedup to 5X. As in both execution times, $FPGA_{full}$ and $FPGA_{post}$, the communication is completely included the software turns out to be the bottleneck.

Query 2 (Figure 10) has the biggest result set of all test queries. The speedup of the hybrid system is slightly increasing up to 5X faster with a growing dataset size.

Due to the enormous result size the post processing on the host further shrinks the speedup. Besides the enormous result size, the union operator causes the speedup degradation. Although this operator is very simple it tends to consume one intermediate result of one preceding operator and stalls the other preceding operator. In fact, it stalls a whole branch including a join in this particular query.

Query 3 (Figure 11) contains 3 joins and has significantly less results than Query 2, but more than Query 1. Due to the previously described higher amount of concurrent operators the hybrid system is able to show steady speedup improvements of up to 28 time faster. Again, post processing on the host shrinks the achieved speedup significantly down to a still significant speedup of 5.

Query 4 is an extension of Query 3 by introducing a new variable and triple pattern resulting in another join. The number of resulting bindings array stays the same, but notice that regarding Query 4 each bindings array contains one more variable causing a 25% higher bandwidth need and post processing overhead. Due to the additional join, the software-only execution needs more time to evaluate the query (Figure 12). Contrary, the FPGA-accelerated execution shows almost no performance drop because the additional join lies in another branch of the operator graph and is perfectly integrated into the operator pipeline. As a result the speedup rises up to 32. However, at some point the speedup drops slightly but increases again. Contrary, the host system is not able to counterbalance this drop and post processing shrinks the speedup down to 5.

Query 5 further extends Query 4 by another join respectively variable. This time the query causes a significantly smaller results set than the other queries. In Figure 13, it can be nicely seen that also this query suffers a speedup drop at 66M but afterwards stabilizes and increases. As the result is relatively small the post processing on the host does not have a negative effect on the execution time.

It is worth to stress again the fact that the reported execution times and achieved speedups include the whole communication between host and FPGA which also cover (i) reading triples at host side from hard disk, (ii) sending triples to the FPGA using PCIe and (iii) on the other hand sending back the result from FPGA to host and iterate through it on host side.

5.3 Billion Triples Challenge

In order to address real-world scenarios we imported the Billion Triples Challenge (BTC) dataset [24] which is a crawl of multiple sources (such as *DBpedia* and *freebase*) containing more than one billion distinct triples

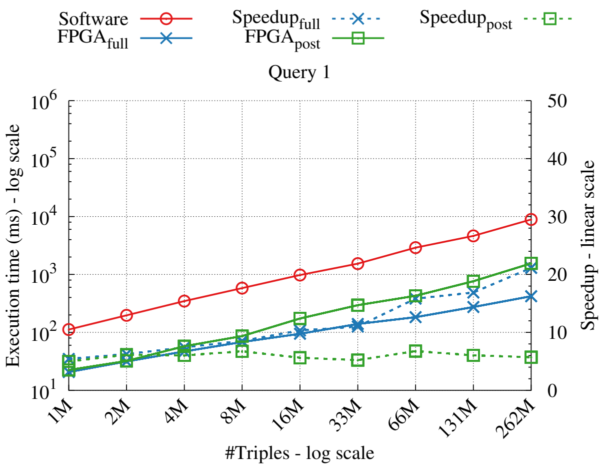

Figure 9: Execution time of Query 1 for different dataset sizes

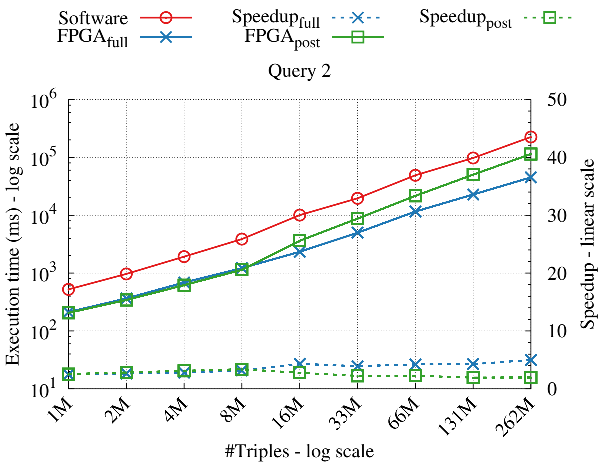

Figure 10: Execution time of Query 2 for different dataset sizes

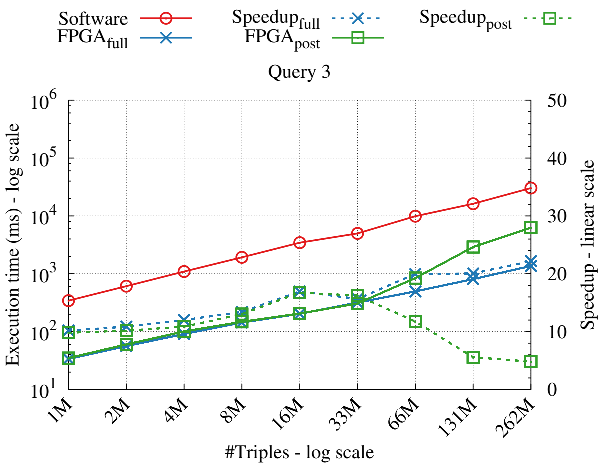

Figure 11: Execution time of Query 3 for different dataset sizes

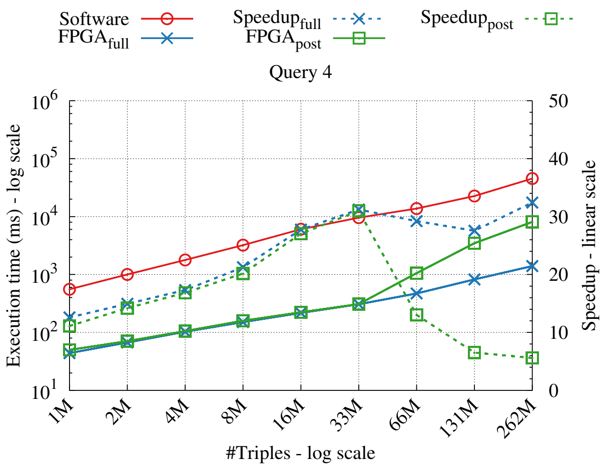

Figure 12: Execution time of Query 4 for different dataset sizes

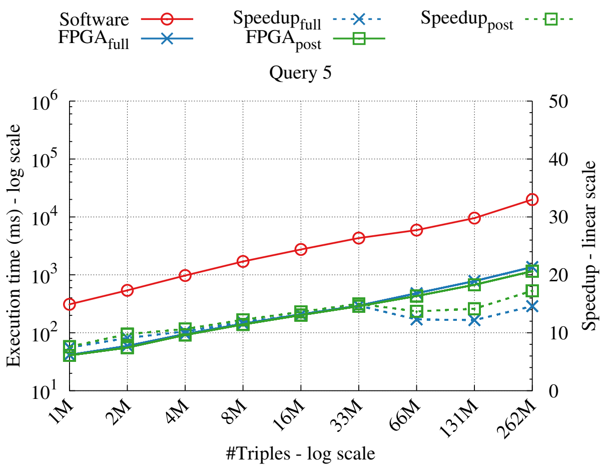

Figure 13: Execution time of Query 5 for different dataset sizes

(1,056,184,909 without duplicates). The dataset does not provide any reference queries. Therefore, we chose 10 queries with different complexities regarding amount of operators, join distribution and result sizes (BTC-1 to BTC-10). The actual queries can be found in the appendix. Figure 14 shows the resulting execution times for software-only and FPGA (full, post) as well as the corresponding speedup. It can be seen that the FPGA-accelerated approach is often slightly faster (except Query 2) and many times significantly outperforms the software-only approach. Again, all reported execution times include the communication costs between host and FPGA.

Although BTC-1 is a very simple query, consisting of only one join, the hybrid system outperforms the software-only approach. BTC-2 is an extension of BTC-1 with an additional triple pattern and join, but the

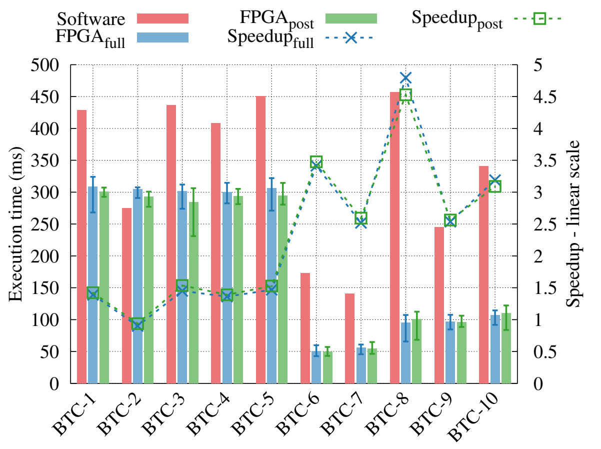

Figure 14: Execution time of test queries on the BTC-2012 dataset [24]

new pattern matches only a small number of triples. Thus, one join causes only low workload and the query execution does not benefit from the FPGA. However, the performance loss is almost not notable. In turn, BTC-3 has the same query structure as BTC-2, but the triple patterns match more triples, and this results in a higher utilization of the joins which has no effect on the hybrid system but on the software-only approach due to higher workload.

BTC-4 and BTC-5 further extend BTC-3 by one respectively two additional triple patterns (with low amount of matches) resulting in one respectively two additional joins. The latter causes higher workload on the software-only approach but has no impact on the hybrid system. BTC-6 has the same structure as BTC-5, but the distribution of triples between index scans is more homogenous. Further, the involved joins generate less intermediate results, and this results in a significantly lower execution time than in the previous queries. BTC-7 extends BTC-6 by another triple pattern and join resulting in a 10 times smaller result, and this is beneficial for the software-only approach but still slower than the hybrid system.

BTC-8 results in a perfectly balanced operator graph consisting of four index scans and three joins which enables the hybrid system to make use of its inherent advantages. BTC-9 adds another triple pattern without any impact on the hybrid system but performance gain of software-only system. BTC-10 extends BTC-9 by adding two triple patterns resulting in two additional joins. Again, while the execution time of the hybrid system slightly changes, the performance of software-only approach degrades due to the additional operators. In summary, it can be seen that the hybrid system is less sensible to the query structure but also the post processing on the host system is negligible.

6 Future Work and Summary

In the following we point out one remaining issue of the presented architecture and sketch two possible solutions, which will be investigated in the future. Finally, we summarize this work.

6.1 Partition Granularity and Interconnects

One important aspect is not yet addressed in the performance evaluation. Due to complexity of the place and routing process, which maps the query to actual resources on the FPGA, the bitfile generation takes between 20 to 30 minutes. Thus, although fully functional the system is not efficiently applicable in a real-world scenario. A straightforward approach is the generation of bitfiles of frequently issued queries and reuse them during runtime. In fact, the detection and reuse of known queries is already prototypically implemented in the proposed system. However, in a highly dynamic environment this approach is not applicable as any arbitrary query is possible.

Therefore, we can divide the chip area into multiple dynamic partitions (tiles). Each tile is able to take one (but arbitrary) operator. Thus each tile should provide the same resources. An efficient way for identification of homogenous reconfiguration areas is presented by Backasch et al. [1]. During system design time the partial bitfile for each operator and each possible tile needs to be generated. In order to adjust operator specific properties (such as position of join variable) it is necessary to store the operator's parameters in registers within the operator and modify their content by manipulating the bitfile according to the query. Another challenge is to establish a flexible interconnection between the partitions. In the following we discuss two approaches to overcome this problem in the future.

Semi-static Operator Graph: In a static operator graph a template of a general query structure without actual operators would be pre-configured on the static design. Obviously, this would significantly reduce the number of possible queries because the connections between two operators can not be changed. Adding additional switching resources (with multiple predecessor and successors), which can be triggered from within an operator, reduces this problem. As the interconnection can be modified during system runtime by exchanging operator bitfiles, we call this approach Semi-static Operator Graph. However, depending on the complexity of the switching resources the possible graph structure is still limited.

NoC-based Interconnect: Backasch et al. [2] presented a generic hardware design which allows the composition of application specific data paths at system runtime. The interconnects between different tiles is realized by using a Network on Chip (NoC) and allows not only the communication between neighboring tiles but to any other tile in the NoC. As this introduces delays in the data propagation it is still desirable that neighbors in the operator graph are placed beside each other in the NoC. However, this approach enables a high degree of flexibility in placing one or more operator graphs onto an FPGA.

6.2 Conclusion

In this paper, we presented the first fully integrated hardware-accelerated query engine in the context of Semantic Web databases. An FPGA is used as an runtime reconfigurable accelerator to flexibly address the variety of SPARQL queries. The dynamic partition in the FPGA is automatically assembled by using a query template and a pool of operators. With respect to the query the contained operators are connected with each other and parametrized by operator specific generics. As all operators provide a common interface, the presented framework can be easily extended by new operator implementations. Furthermore, we executed several queries on large-scale synthetic and real-world data from the Billion Triples Challenge to show the architecture's feasibility and potential to speedup query execution significantly.

However, it was shown that not all queries remarkably benefit from the hardware-accelerated execution and thus it is reasonable to develop an estimator, which is able to predict the expected performance gain. At this stage, the host systems holds all the initial data to be queried. As our FPGA is equipped with a SATA interface it might be reasonable to store triple data at hard drives attached on the FPGA. Consequently, the communication overhead and the load on the host will be significantly reduced because only the result of a query needs to be sent to and processed on the host system. Additionally, we intend to take advantage of FPGAs in other computationally intensive database tasks such as index generation as well.

Acknowledgements

This work is funded by the German Research Foundation (DFG) project GR 3435/9-1.

References

[1] R. Backasch, G. Hempel, S. Groppe, S. Werner, and T. Pionteck, "Identifying Homogenous Reconfigurable Regions in Heterogeneous FPGAs for Module Relocation," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 2014.

[2] R. Backasch, G. Hempel, T. Pionteck, S. Groppe, and S. Werner, "An Architectural Template for Composing Application Specific Datapaths at Runtime," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 2015.

[3] A. Becher, F. Bauer, D. Ziener, and J. Teich, "Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL 2014)*. IEEE, 2014, pp. 662–669.

[4] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, May 2001.

[5] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[6] S. Breß, "Efficient Query Processing in Co-Processor-accelerated Databases," Ph.D. dissertation, Otto-von-Guericke-Universität Magdeburg, 2015.

[7] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake, "Ocelot/hype: Optimized data processing on heterogeneous hardware," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1609–1612, Aug. 2014.

[8] S. Breßand G. Saake, "Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1398–1403, Aug. 2013.

[9] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 151–160.

[10] X. Cheng, B. He, and C. T. Lau, "Energy-efficient query processing on embedded cpu-gpu architectures," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, ser. DaMoN'15. New York, NY, USA: ACM, 2015, pp. 10:1–10:7.

[11] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, vol. 20, pp. 45–52, 2012.

[12] C. Dennl, D. Ziener, and J. Teich, "Acceleration of SQL Restrictions and Aggregations Through FPGA-Based Dynamic Partial Reconfiguration," in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 25–28.

[13] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The gamma database machine project," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, no. 1, pp. 44–62, Mar. 1990.

[14] D. DeWitt, "Direct - a multiprocessor organization for supporting relational database management systems," *Computers, IEEE Transactions on*, vol. C-28, no. 6, pp. 395–406, June 1979.

[15] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1221–1230.

[16] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 670–680, Sep. 2010.

[17] J. Gantz and D. Reinsel, "Te Digital Universe in 2020: Big Data, Bigger Digital Shadow s, and Biggest Grow th in the Far East," http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, December 2012, accessed: 2016-05-05.

[18] G. Giannikis, G. Alonso, and D. Kossmann, "Shareddb: Killing one thousand queries with one stone," *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 526–537, Feb. 2012.

[19] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: High performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 325–336.

[20] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, Jun. 1993.

[21] S. Groppe, "LUPOSDATE Open Source," https://github.com/luposdate, 2013.

[22] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer Verlag, Heidelberg, 2011. [Online]. Available: http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/

[23] S. A. Guccione, "Chapter 3 - reconfigurable computing systems," in *Reconfigurable Computing*, ser. Systems on Silicon, S. Hauck and A. Dehon, Eds. Burlington: Morgan Kaufmann, 2008, pp. 47–64.

[24] A. Harth, "Billion Triples Challenge data set," Downloaded from http://km.aifb.kit.edu/projects/btc-2012/, 2012.

[25] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 21:1–21:39, Dec. 2009.

[26] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 511–524.

[27] B. He and J. X. Yu, "High-throughput transaction executions on graphics processors," *Proc. VLDB Endow.*, vol. 4, no. 5, pp. 314–325, Feb. 2011.

[28] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled cpu-gpu architectures," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 329–340, Dec. 2014.

[29] D. Heinrich, S. Werner, M. Stelzner, C. Blochwitz, T. Pionteck, and S. Groppe, "Hybrid FPGA Approach for a B+ Tree in a Semantic Web Database System," in *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*. Bremen, Germany: IEEE, Jun. 2015.

[30] IBM Corp., "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics," IBM, 2011.

[31] Z. Istvan, L. Woods, and G. Alonso, "Histograms as a side effect of data movement for big data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1567–1578.

[32] Y. Kang, Y. suk Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST 13)*, May 2013.

[33] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, vol. 27, no. 3, pp. 42–52, Sep. 1998.

[34] D. Koch and J. Torresen, "Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 45–54.

[35] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory ssd technology for enterprise database applications," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 863–870.

[36] H.-O. Leilich, G. Stiege, and H. C. Zeidler, "A search processor for data base management systems," in *Proceedings of the fourth international conference on Very Large Data Bases - Volume 4*, ser. VLDB'1978. VLDB Endowment, 1978, pp. 280–287.

[37] Linked Open Data, "Connect Distributed Data across the Web." [Online]. Available: http://linkeddata.org/

[38] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: Memory access," *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, Dec. 2000.

[39] C. Meenderinck and B. H. H. Juurlink, "(When) Will CMPs Hit the Power Wall?" in *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, 2008, pp. 184–193.

[40] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires: A Query Compiler for FPGAs," *Proc. VLDB Endow.*, vol. 2, pp. 229–240, August 2009.

[41] R. Mueller, J. Teubner, and G. Alonso, "Glacier: A Query-to-Hardware Compiler," in *Proceedings of the 2010 International Conference on Management*

of Data, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1159–1162.

[42] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Flexible Query Processor on FPGAs," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1310–1313, Aug. 2013.

[43] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "The FQP Vision: Flexible Query Processing on a Reconfigurable Computing Fabric," *SIGMOD Rec.*, vol. 44, no. 2, pp. 5–10, Aug. 2015.

[44] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in fpga systems: A survey and a cost model," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, 2011.

[45] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen, "Multi-query Stream Processing on FPGAs," in *ICDE*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 1229–1232.

[46] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench," http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/download.php, 2009.

[47] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL Performance Benchmark," in *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009, pp. 222–233.

[48] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 67–80.

[49] A. Shatdal, C. Kant, and J. F. Naughton, "Cache conscious algorithms for relational query processing," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 510–521.

[50] C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration for spatial selections and joins," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 455–466.

[51] J. Teubner and R. Mueller, "How Soccer Players Would do Stream Joins," in *Proceedings of the 2011 International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 625–636.

[52] The W3C SPARQL Working Group, "SPARQL 1.1 Overview," https://www.w3.org/TR/sparql11-overview/, 2013, W3C Recommendation.

[53] S. Werner, S. Groppe, V. Linnemann, and T. Pionteck, "Hardware-accelerated Join Processing in Large Semantic Web Databases with FPGAs," in *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS 2013)*. Helsinki, Finland: IEEE, Jul. 2013, pp. 131–138.

[54] S. Werner, D. Heinrich, J. Piper, S. Groppe, R. Backasch, C. Blochwitz, and T. Pionteck, "Automated Composition and Execution of Hardware-accelerated Operator Graphs," in *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*. Bremen, Germany: IEEE, Jun. 2015.

[55] S. Werner, D. Heinrich, M. Stelzner, S. Groppe, R. Backasch, and T. Pionteck, "Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases," in *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT2014)*. Xi'an, China: IEEE, Sep. 2014, pp. 539–546.

[56] S. Werner, D. Heinrich, M. Stelzner, V. Linnemann, T. Pionteck, and S. Groppe, "Accelerated join evaluation in Semantic Web databases by using FPGAs," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 2031–2051, May 2015. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cpe.3502/abstract

[57] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 963–974, Jul. 2014.

[58] L. Woods, J. Teubner, and G. Alonso, "Less watts, more performance: An intelligent storage engine for data appliances," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1073–1076.

[59] World Wide Web Consortium (W3C), "RDF 1.1 Concepts and Abstract Syntax," https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014, W3C Recommendation.

[60] Xilinx, "Virtex-6 Family Overview," http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, January 2012, DS150 (v2.5).

[61] Xilinx, "Virtex-6 FPGA CLB - User Guide," February 2012.

[62] Xilinx, "Partial Reconfiguration User Guide," http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/ug702.pdf, April 2013, UG702 (v14.5).

[63] Xillybus Ltd., "Xillybus Website," http://xillybus.com, May 2016, accessed: 2016-05-05.

APPENDICES

Used prefixes

```
PREFIX bench: <http://localhost/vocabulary/
    bench/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf
    -syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
    schema#>
PREFIX swrc: <http://swrc.ontoware.org/
    ontology#>
```

Query 1

```
#Get all articles with property swrc:pages.
SELECT ?article
WHERE { ?article rdf:type bench:Article .
        ?article ?property ?value .
        FILTER (?property=swrc:pages)}
```

Query 2

```
#Get incoming and outcoming properties of
    persons.
SELECT ?predicate
WHERE {
  { ?person rdf:type foaf:Person .
    ?subject ?predicate ?person
  } UNION {
    ?person rdf:type foaf:Person .
    ?person ?predicate ?object
  }
}
```

Query 3

```
#Get all articles with title, number of pages
    and creator.
SELECT ?article ?title ?pages ?creator
WHERE { ?article rdf:type bench:Article .
        ?article dc:title ?title .
        ?article swrc:pages ?pages .
        ?article dc:creator ?creator }
```

Query 4

```
#Get all articles with titles, number of
    pages, the creator and the journal where
    published.
SELECT ?article ?title ?pages
       ?creator ?journal
WHERE { ?article rdf:type bench:Article .
        ?article dc:title ?title .
        ?article swrc:pages ?pages .
        ?article dc:creator ?creator .
        ?article swrc:journal ?journal}
```

Query 5

```
#Get all articles with titles, number of
    pages, the creator, journal and month
    when published.
SELECT ?article ?title ?pages
       ?creator ?journal ?month
WHERE { ?article rdf:type bench:Article .
        ?article dc:title ?title .
        ?article swrc:pages ?pages .
        ?article dc:creator ?creator .
        ?article swrc:journal ?journal .
        ?article swrc:month ?month}
```

BTC-1

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbp:name ?title}
```

BTC-2

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbp:name ?title.
  ?book1 dbp:pubDate ?date}
```

BTC-3

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbp:name ?title.
  ?book1 dbp:country ?country}
```

BTC-4

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbp:name ?title.
  ?book1 dbp:country ?country.
  ?book1 dbp:pages ?pages}
```

BTC-5

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbp:name ?title.
  ?book1 dbp:country ?country.
  ?book1 dbp:pages ?pages.
  ?book1 rdf:type dbo:Book}
```

BTC-6

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbo:isbn ?isbn.
  ?book1 dbp:country ?country.
  ?book1 dbp:pages ?pages.
  ?book1 rdf:type dbo:Book}
```

BTC-7

```
select * where {
  ?book1 dbp:author ?author.
  ?book1 dbo:isbn ?isbn.
  ?book1 dbp:country ?country.
  ?book1 dbp:pages ?pages.
  ?book1 rdf:type dbo:Book.
  ?book1 dbp:pubDate ?date.}
```

BTC-8

```
select * where {
  ?s dbp:countryofbirth ?o1 .
  ?s dbo:birthDate ?o3 .
  ?s rdf:type dbo:Athlete .
  ?s dbp:fullname ?o5}
```

BTC-9

```
select * where {
  ?s dbp:countryofbirth ?o1 .
  ?s dbp:countryofdeath ?o2 .
  ?s dbo:birthDate ?o3 .
  ?s dbo:deathDate ?o4 .
  ?s rdf:type dbo:Athlete}
```

BTC-10

```
select * where {
  ?s rdf:type dbo:Athlete .
  ?s dbp:countryofbirth ?o1 .
  ?s dbp:countryofdeath ?o2 .
  ?s dbo:birthDate ?o3 .
  ?s dbo:deathDate ?o4 .
  ?s dbp:fullname ?o5}
```

AUTHOR BIOGRAPHIES

Stefan Werner received his Dipl.-Inf. degree in Computer Science in March 2011 at the University of Lübeck, Germany. Now he is a research assistant and PhD candidate at the Institute of Information Systems at the University of Lübeck. His research focuses on multi-query optimization and the integration of a hardware accelerator for relational databases by using run-time reconfigurable FPGAs.

Dennis Heinrich received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. At the moment he is employed as a research assistant at the Institute of Information Systems at the University of Lübeck. His research interests include FPGAs and corresponding hardware acceleration possibilities for Semantic Web databases.

Sven Groppe earned his diploma degree in Informatik (Computer Science) in 2002 and his Doctor degree in 2005 from the University of Paderborn. He earned his habilitation degree in 2011 from the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, an open-source Semantic Web database, and one of the project leaders of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. His research interests include databases, Semantic Web, query and rule processing and optimization, Cloud Computing, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Christopher Blochwitz received his M.Sc. in Computer Science in September 2014 at the University of Lübeck, Germany. Now he is a research assistant/ PhD student at the Institute of Computer Engineering at the University of Lübeck. His research focuses on hardware acceleration, hardware optimized data structures, and partial reconfiguration of FPGAs.

Thilo Pionteck is head of the chair of “Hardware-Oriented Technical Computer Science” at the Otto von Guericke University Magdeburg, Germany. He received his Diploma degree in 1999 and his Ph.D. (Dr.-Ing.) degree in Electrical Engineering both from the Technische Universität Darmstadt, Germany. In 2008 he was appointed as an assistant professor for “Integrated Circuits and Systems” at the Universität zu Lübeck. From 2012 to 2014 he was substitute of the chair of “Embedded Systems” at the Technische Universitt Dresden and of the chair of “Computer Engineering” at the Technische Universität Hamburg-Harburg. Before moving to Magdeburg in 2016 he was an associate professor for “Organic Computing” at the Universität zu Lübeck. His research work focus on adaptive system design, runtime reconfiguration, hardware/software codesign and network-on-chips.