Datenbank Spektrum (2021) 21:55–64
https://doi.org/10.1007/s13222-020-00363-7

SCHWERPUNKTBEITRAG

Speculative Dynamic Reconfiguration and Table Prefetching Using Query Look-Ahead in the ReProVide Near-Data-Processing System

Lekshmi Beena Gopalakrishnan[1] · **Andreas Becher**[2] · **Stefan Wildermann**[2] · **Klaus Meyer-Wegener**[1] · **Jürgen Teich**[2]

Received: 30 May 2020 / Accepted: 14 December 2020 / Published online: 4 January 2021
© The Author(s) 2020

Abstract
FPGAs are promising target architectures for hardware acceleration of database query processing, as they combine the performance of hardware with the programmability of software. In particular, they are partially reconfigurable at runtime, which allows for the runtime adaptation to a variety of queries. However, reconfiguration costs some time, and a region of the FPGA is not available for computations during its reconfiguration. Techniques to avoid or at least hide the reconfiguration latencies can improve the overall performance. This paper presents optimizations based on query look-ahead, which follows the idea of exploiting knowledge about subsequent queries for scheduling the reconfigurations such that their overhead is minimized. We evaluate our optimizations with a calibrated model for various parameter settings. Improvements in execution time can be "calculated" even if only being able to look one query ahead.

Keywords Query optimization · Hardware acceleration · Reconfiguration · FPGAs

1 Introduction

Some research has been conducted to address the acceleration of database query processing with the help of FPGAs, e.g. [16, 21, 27], and its integration into a DBMS. The ReProVide (for "Reconfigurable data ProVider") approach [5] is part of it. It introduces a storage-attached hardware technology called ReProVide Processing Unit (RPU) to process queries completely or at least partially close to the data source, in synergy with a DBMS. This co-design approach exploits dynamic reconfiguration of FPGAs in the RPUs in combination with a novel DBMS optimizer. On the RPUs, a library of query-processing modules is stored, which can be configured onto the FPGA in a few milliseconds. However, due to the limited amount of logic resources on an FPGA, not all modules can be the resident simultaneously. So at each point in time, only a subset is ready for use in query processing. Hence, *reconfiguration* of one or more regions of the FPGA is needed.

This paper discusses the idea of exploiting the knowledge of *sequences of queries* to reduce the number of these reconfigurations. Information about the sequences can be given to the RPU via so-called *hints*. It is shown how an RPU can use this information to reduce the overall execution time for the sequence of queries. We will indicate where such sequences of queries can be found. They are, e.g., part of an application program that fills the frames of a modular screen output from different records of an underlying database.

The basic idea and preliminary evaluations were initially published in [7] and [6]. This paper adds new optimizations and provides a series of new evaluations that make the effect of the optimizations much clearer.

The paper is structured as follows: In Section 2, we introduce the concepts of the ReProVide approach and its RPUs. Other approaches to hardware acceleration and the matching query optimization are identified in Section 3. Important information and assumptions on query sequences and their processing are detailed in Section 4, while Section 5 describes the execution of queries involving an RPU and the proposed solutions to optimize query-sequence execution with a focus on techniques to eliminate or hide FPGA reconfiguration times. A detailed evaluation model of execu-

✉ Lekshmi Beena Gopalakrishnan
lekshmi.bg.nair@fau.de

✉ Andreas Becher
andreas.becher@fau.de

[1] INF6, FAU Erlangen-Nürnberg, Erlangen, Germany

[2] INF12, FAU Erlangen-Nürnberg, Erlangen, Germany

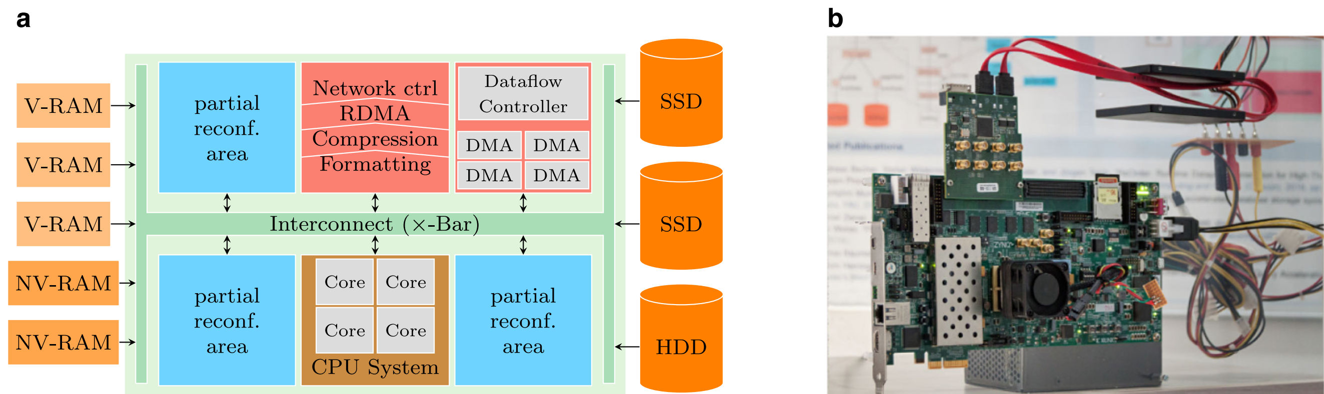

Fig. 1 **a** Architecture of a ReProVide Processing Unit (RPU). **b** RPU Prototype [8]

tion times is then derived in Section 6. Section 7 presents evaluations of the presented concepts, and Section 8 discusses strategies for future optimizations of reconfigurable systems based on query sequences. Finally, Section 9 concludes the paper.

2 ReProVide

The goal of the ReProVide approach [5] is to investigate FPGA-based solutions for smart storage and near-data processing together with novel query-optimization techniques that exploit the speed and reconfigurability of FPGA hardware for a scalable and powerful (pre-)filtering of Big Data. Our ReProVide prototype was presented first in [8]. Figure 1 shows the logical concept of an RPU and a photograph of the operational physical platform. For this storage-attached device, direct interfaces to a multitude of memory and storage types have been designed and implemented. Our investigations in this paper use SSDs, where data tuples are stored row-wise. An RPU exploits the capabilities of dynamic (runtime) hardware reconfiguration of modern FPGAs to load pre-designed hardware accelerators on-the-fly. It can process SQL queries or parts of them in hardware in combination with the CPU cores also available on the PSoC.

The FPGA of an RPU contains various static hardware modules, such as a storage controller, a network controller, data interconnects, and local memories, in addition to multiple partially reconfigurable regions (PRs) shown in blue in Fig. 1a. Data is processed by so-called *accelerators* loaded into these PRs. RPUs are able to either process a query in full or at least partially by streaming the tables from the storage at line-rate through one or many of these accelerators to the network interface. It is important to note that streaming a table from an attached storage through the FPGA comes at no additional cost[1]. However, some operations like sorting or joining larger tables can be processed at higher speed by the DBMS. As hardware accelerators are best for line-rate processing, and FPGA resources are limited, not all available operator modules can be combined into a single accelerator. E.g., implementations of arithmetical operations (multiplication, addition, ...) depend on the data type they operate on (float, int32, int64). Via dynamic partial reconfiguration, it is possible to spatially multiplex more operators on the same platform. Currently, the operators for filtering, projection, and semi-join are supported in hardware, see [5], handling integers as well as strings and floats. The set is growing continuously. Attribute values can be used in arithmetical calculations, before they are compared with constants or with each other. The comparisons supported are the usual θ operations, i.e., $<, >, =, \neq, \leq, \geq$.

Each RPU configuration consists of a pipeline of concatenated, partially reconfigurable accelerator modules. For instantiating them in one or multiple PRs, some additional cost (time) for reconfiguration will be encountered.

The complexity of a filter-predicate evaluation only influences the hardware requirements of the accelerator. For example, if we have an accelerator to evaluate predicate conditions given in disjunctive normal form (DNF) and consisting of n clauses, then this accelerator can also evaluate predicates given in DNF with $1, 2, 3, ..., n-1$ clauses. In all cases, the throughput rate of the accelerator remains the same. The complexity of the predicate evaluation does not influence time and throughput, only the cardinality of the input data does. If the predicate is too complex for the accelerator (e.g. the number of clauses is greater than n), then it cannot be mapped onto the accelerator at all. This is handled by what we call capability-aware query optimization—see below.

An RPU is connected to a host via a fast network. At this host a (relational) DBMS is running, executing queries

[1] We assume that the data rate of the network is smaller than the data rate of the RPU storage.

that access the tables on the RPU and combine them with data from other tables stored at the host itself. For the integration of RPUs into a DBMS, we have elaborated a novel hierarchical (multi-level) query-optimization technique to determine which operations are worthwhile to be assigned to an RPU (query partitioning) and how to deploy and execute the assigned (sub-)queries or database operations at the RPU (query placement). Pushing down query operations to the RPU can greatly reduce the amount of data transferred from the RPU to the host, and thus not only relieves the network of traffic, but also reduces the load on the DBMS. The implemented query optimizer is sharing the work between the global optimizer of the DBMS performing query partitioning and an architecture-specific local optimizer running on the CPU system of the RPU performing query placement.

As demonstrated in [8] our query optimizer has been extended with our own cost models and optimization rules, which decide whether to partition a QEP into sub-trees and assign some of them to the RPU based on its capabilities. The information about the capabilities of an RPU (i.e., the set of accelerators available on it) and the local statistics about the data[2] are exchanged via a hint interface [4], which helps the global optimizer in the effective partitioning of a query before push-down. These hints do not change any functionality, but give information to the RPU that allows it to optimize the details of processing a query on the RPU. This paper underlines the significance of such hints in avoiding unnecessary reconfigurations or beginning a reconfiguration for forthcoming queries while still executing the current one.

3 Related Work

Query-sequence optimization was addressed already in [9, 18, 23] quite a while ago. They focused on the optimization of a sequence of SQL statements generated by a ROLAP query generator. The coarse-grained optimization (CGO) they introduced has optimization strategies that include rewriting the queries to combine them, using a common `WHERE` clause, `HAVING` clause, or `SELECT` clause, and to partition them, enabling parallel execution. The decision about which optimization strategy to use is only based on information about the query sequence itself, which means they start to optimize the sequence once the whole sequence is known. A query sequence is of interest to us if it appears frequently in the query log [34]. The structure of the query, however, is important for us. Hardware acceleration has not been considered in any of the mentioned projects.

Query processing using new hardware has been studied extensively in recent years [1]. The self-tuning optimizer framework HyPE (Hybrid Query Processing Engine) [10–13] allows hybrid query processing. It has been used to build CoGaDB's optimizer, which efficiently distributes a workload to available processors. It is hardware- and algorithm-oblivious, i.e., it has only minimal knowledge of the underlying processors or the implementation details of operations. CoGaDB also caches frequently used columns (in GPU memory) based on query patterns observed in the past. This is a form of multi-query optimization, too. In contrast to this, our system uses the queries and/or features of a sequence (such as tables, attributes, and operations used) based on previous sequence patterns.

The ADAMANT project described in [4] also enhances a DBMS with extensible, adaptable support for heterogeneous hardware. The main objectives of this project are to consider the fundamental differences between the heterogeneous accelerators, and the development of an improved optimization process that copes with the explosion of the search space. They also exploit the different features available in heterogeneous co-processors, mainly emphasizing GPUs, but also looking into FPGAs.

Offloading operations for query processing to FPGA-based hardware accelerators has been researched well in [2, 3, 20, 25, 28, 31, 35], because of its small energy footprint and fast execution. Next to traditional co-processor systems, the "bump-in-the-wire" approach (as applied in ReProVide) has also been of great interest in approaches described in [16, 28, 32]. The other projects related to FPGAs that we are aware of are all focusing on the execution of a single query. We extend this with the execution of a sequence of queries to reduce the number of reconfigurations.

Multi-query optimization, as investigated e.g. in [14, 15, 24], can also be used in our approach, because query-result caching and materialized views are good choices and widely adopted techniques to improve query processing. Our technique is tailored to query sequences, which originate in a setup where a given set of applications is executed in response to user interactions (e.g., in a Web shop). Each application contains multiple queries, which are invoked sequentially, and between the queries other program code is executed. Usually, the parameters of the queries are set to user-specific values. In such a setup, our techniques are actually well suited to complement query-result caching or materialized views. They are expected to work perfectly for the static (non-user-specific) parts of the queries and serve two purposes: First, they can greatly reduce the size of the original datasets. And second, these reduced datasets may even fit into main memory. Both improves the effect of our optimizations significantly, as our evaluation will show. Our

[2] Cardinality estimates can be refined each time a query is executed, as the RPU can generate statistics about the data without additional delay, which fine-tune our cost model and thus allow better choices of QEPs.

technique is tailored to those parts of reappearing queries that contain user-specific parameter values, which differ in subsequent executions of the applications and their queries (e.g., user id, product category, etc. in a Web shop). Results cannot be materialized or cached for each possible user id, etc. Still, our technique benefits from caching and materialized views due to the aforementioned effects of dataset reduction and faster access to main memory.

A much more elaborated scenario can be found in the Tableau software accessing Hyper [29] and in some other commercial tools.

4 Query-Sequence Model

Query sequences can be obtained from database query logs as in [17, 22, 30, 33], for instance, or by code analysis as in [19, 26], including the frequency and the time of arrival. More formally, a query sequence is an ordered set of n individual queries $Q_0, ..., Q_{n-1}$. The time gap t_{gap,Q_i}, with $i \in [0...n-2]$, between two successive queries (from the co mpletion of Q_i to the arrival of Q_{i+1}) is estimated from th e observations, e.g. by taking the average. It is expected to be in the order of a few milliseconds, but that of course de pends on the code of the application program.

The optimizer of the DBMS analyzes the query sequences (similarly to [15]) to determine the tables and attributes accessed, and in which order, as well as the operations used on the attributes of the tables. This information can be extracted and organized with the help of the query repository [22]. The goal is to accelerate analytical queries, that is, read-only queries with large scans. While the proposed approach is suitable for arbitrary operations, we focus on filters (selections) together with arithmetic for now in our examples. The values used in the arithmetic or in the comparisons may come from program variables, so they may vary in different executions of the query sequence. We will introduce parameters here, as known from prepared queries and stored procedures[3].

When a new query is received and handed to the optimizer, it will be optimized in the usual way, including query partitioning and query placement as described in Section 2. In addition to that, the optimizer searches the repository for sequences with this query as the first. If a sequence can be found, the optimizer may (a) reconsider the query partitioning and (b) pass the information about the queries assumed to follow (or even the complete sequence) as hints to the RPU, allowing it to optimize even further[4]. The RPU should then try to avoid reconfigurations and thus reduce the execution time of the pushed-down query plans.

5 Execution Model and Optimizations

To focus on the reconfiguration and to keep the model close to the prototype, we consider only a single PR of the RPU. This means that processing phases with dependencies, such as loading and executing an accelerator, are not done in parallel. However, as the RPU uses hardware to implement various functions, it embodies a dedicated circuit for table scans. This allows a table scan to run in parallel with other operations at all times. The target of this model is not to optimize the execution time of a single query, but the time of the whole sequence, that is, the time from the arrival of Q_0 until the transfer of the last result of Q_{n-1}, including all time gaps.

Figure 2 shows possible execution strategies for a query sequence with two consecutive queries (Q_0, Q_1). This is a minimum-size sequence. Any other sequence with more queries (and thus more accelerator runs) will only enhance the possibility to apply the optimizations presented here, since more than one consecutive pair of queries is available to find similarities and to generate hints from them. The assumption here is that the second query uses an operation that can be executed by the accelerator acc_0, and this accelerator is loaded on the RPU already, because it was needed for one of the operations of the first query. We further assume that the two operations of Q_0 commute.

The standard plan **S** is generated by an optimizer that looks at each query individually and does not take any hints into account. It only considers selectivities to order the filter predicates.

The symbols in the standard plan **S** of Fig. 2 have the following meaning: t_S is the total execution time for the sequence. The execution times t_{Q_i} of the queries comprise the reconfiguration times (t_r), the table-scanning times $(t_{Q_i,scan})$, the accelerator-execution times (t_{S,Q_0,acc_0}, t_{S,Q_0,acc_1}, t_{Q_1,acc_0}) and the network-transport times ($t_{S,Q_0,trans}$, $t_{Q_1,trans}$). As the table scan can run while the first accelerator is being loaded, only the maximum of $t_{Q_i,scan}$ and t_r contributes to the execution time of the query[5]. This plan is chosen by the standard optimization not taking query sequences into account. Since the two operations commute, the selectivity factor f_{acc_0} of the first accelerator must be

[3] In fact the query sequences may be regarded as stored procedures identified by the DBMS itself. So they can be optimized prior to the actual invocation.

[4] Of course, query sequences may reappear with variations. This is a bit like the branch prediction in processor design. If one of the next queries is different from that in the found query sequence, again the RPU must be informed via a hint.

[5] The name of the plan S is included only if the symbol's value changes in the other plans.

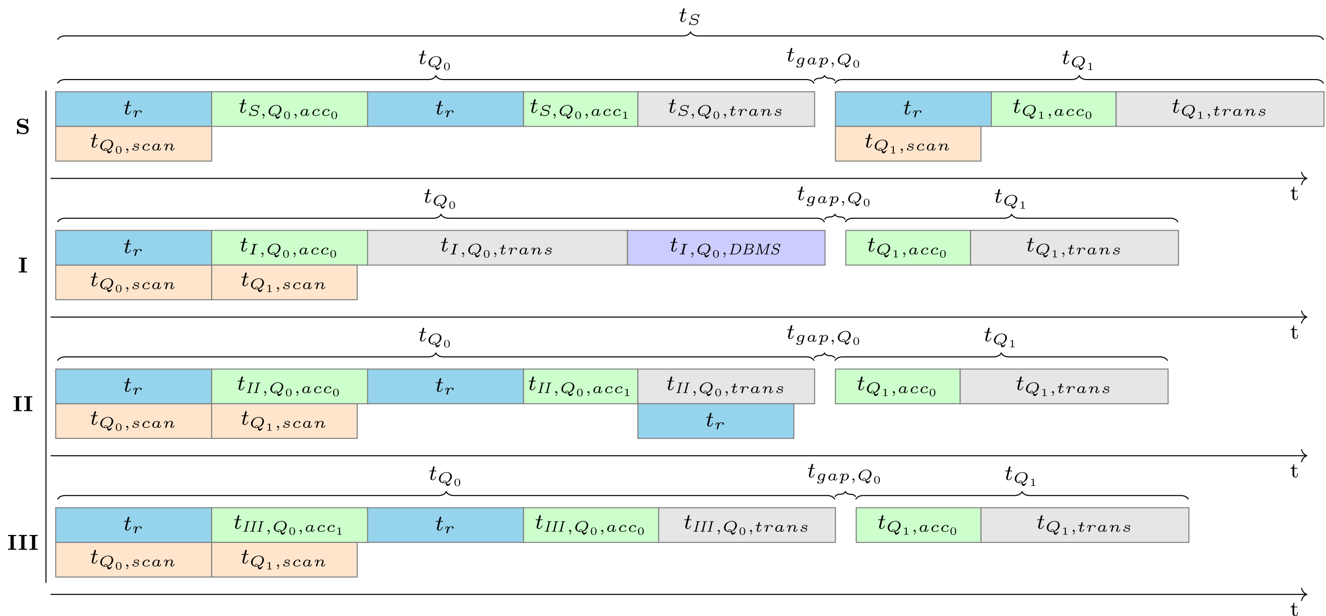

Fig. 2 Bar chart of the processing phases in possible plans for a query sequence (Q_0, Q_1). *Bars* on top of each other show the parallel execution of corresponding operations, i.e., reconfiguration time t_r and table-scanning intervals indexed with *scan*. The standard plan **S** pushes down both operations of Q_0 without taking Q_1 into account or giving any hints. Opt. **I** pushes down only the 1st operation, so the accelerator loaded in the RPU can be reused in Q_1 without reconfiguration. And it gives a hint on the table used in Q_1, so the table scan of Q_1 is started as soon as the resource used for table scans is available again. Opt. **II** uses a further hint about the operations of Q_1 and speculatively loads the accelerator for Q_1, as soon as the 2nd accelerator of Q_0 has finished. Again the table scan for Q_1 starts early. Opt. **III** uses the same hints and swaps the accelerator invocations of Q_0, so that acc_0 becomes 2nd and is already loaded for Q_1. The table scan for Q_1 is again moved upstream

lower than that of the second, otherwise their order would have been the other way around. Furthermore, first executing the filter operations on the RPU and then transferring the result to the DBMS must be more efficient than transferring all the data to the DBMS and applying the filters there. These constraints will be observed when later doing the evaluation of the optimizations.

Optimization strategy **I** in Fig. 2 shows the first proposed optimization to save reconfigurations with the knowledge of the next query in the sequence. The filter operations of the first query are treated differently, such that only the first (with a smaller selectivity factor) is pushed down to the RPU and the second is executed afterwards by the DBMS (which takes the time $t_{I,Q_0,DBMS}$). This avoids reconfiguration twice, for the second accelerator of the first query and for the first operation of query Q_1, but may result in a larger data transfer ($t_{I,Q_0,trans}$).

Alternatively, we can still push down both operations to the RPU, but also give some hints regarding the query coming next. The proposed reconfiguration strategies of the RPU are also shown in Fig. 2. The RPU can then speculatively load accelerators for subsequent queries, once other accelerators are no longer used by the current query (strategy **II**), and avoid the replacement of reusable accelerators by swapping accelerator invocations (strategy **III**). The effect of strategy **II** is that while the processing of the first query Q_0 remains unchanged, the reconfiguration back to the first accelerator (acc_0) is speculatively started as soon as the second accelerator (acc_1) has finished. In the bar chart, the reconfiguration—which runs in parallel with the result transfer—has completed before the subsequent query Q_1 arrives. This is the optimal case for this strategy. Again the accelerator acc_0 of query Q_1 can start processing immediately, without any waiting time introduced by reconfiguration.

Strategy **III** avoids the third reconfiguration by swapping the accelerator invocations of Q_0. As mentioned earlier, it is assumed that the two operations commute. The local optimizer would always invoke the filter with the lowest selectivity first to reduce the data volume early. Thus, swapping may be counterproductive. The lower part of Fig. 2 suggests the consequences: Query Q_0 now needs more time, as the second accelerator acc_0 must process more data and thus takes longer. While this looks detrimental to the goal of acceleration, the third reconfiguration has become obsolete. Thus, the overall execution time for the complete sequence (t_{III}) can still be reduced.

In all three optimization strategies, the hints about the tables and attributes accessed in the forthcoming query can also be used to do the second table scan (in $t_{Q_1,scan}$) prior to receiving the query. This may reduce the execution time

even more, since the data to be processed is already available in memory[6].

6 Evaluation Model

In the following, we provide a "simple, yet sufficient" evaluation model for the strategies shown in Fig. 2. This model is used for evaluating the presented optimization strategies with values determined by measurements on the existing prototype of the RPU. "Please note that a purely selectivity-based predicate ordering is actually all we need for this model."

From the table sizes $s_{Q_i,table}$ of each query Q_i and a scan rate of r_{scan}, we can calculate the scanning time as

$$t_{Q_i,scan} = \frac{s_{Q_i,table}}{r_{scan}}. \tag{1}$$

The reconfiguration time t_r is based on the size of the PRs of our RPU. With a (constant) data rate r_{acc} of the accelerators, we can calculate the processing time of the first accelerator in the standard plan **S** as

$$t_{S,Q_0,acc_0} = \frac{s_{Q_0,table}}{r_{acc}}. \tag{2}$$

According to our assumptions, all our accelerators are filters, so they reduce the size of the table by a selectivity factor of f_{Q_i,acc_j}. This leads to an intermediate table of size

$$s_{S,Q_0,intermediate} = s_{Q_0,table} \times f_{Q_0,acc_0} \tag{3}$$

and a final result size of

$$s_{S,Q_0,result} = s_{S,Q_0,intermediate} \times f_{Q_0,acc_1}. \tag{4}$$

For query Q_1 with only one accelerator in use, we get

$$s_{Q_1,result} = s_{Q_1,table} \times f_{Q_1,acc_0}. \tag{5}$$

This will not change in the other plans. The result of both queries is transferred to the host at a constant network data rate of $r_{network}$, resulting in the transfer times:

$$t_{S,Q_0,trans} = \frac{s_{S,Q_0,result}}{r_{network}} \quad t_{Q_1,trans} = \frac{s_{Q_1,result}}{r_{network}}. \tag{6}$$

[6] It, however, has some consequences. The initial table scan of Q_0 takes place when the query has already arrived. So it can be a full table scan—or an index scan with known values. For the table scan of Q_1 brought forward, the latter is only possible, if the known values are constants within the sequence or variables known prior to Q_1.

Overall, we get an execution time for the standard plan **S** of:

$$t_{S,Q_0} = \max(t_r, t_{Q_0,scan}) + t_{S,Q_0,acc_0} + t_r + t_{S,Q_0,acc_1} + t_{S,Q_0,trans} \tag{7}$$

$$t_{S,Q_1} = \max(t_r, t_{Q_1,scan}) + t_{Q_1,acc_0} + t_{Q_1,trans} \tag{8}$$

$$t_S = t_{S,Q_0} + t_{gap,Q_0} + t_{S,Q_1}. \tag{9}$$

In strategy **I**, we save the reconfiguration time needed for Q_1 and execute the table scan for Q_1 as soon as possible. So the execution time for Q_1 is reduced to

$$t_{I,Q_1} = t_{Q_1,acc_0} + t_{Q_1,trans}. \tag{10}$$

However, a larger transfer time $t_{I,Q_0,trans}$ is needed in Q_0 and also the time $t_{I,Q_0,DBMS}$ for DBMS post-processing. So the calculation changes to:

$$t_I = \max(\max(t_r, t_{Q_0,scan}) + t_{I,Q_0,acc_0} + t_{I,Q_0,trans} + t_{I,Q_0,DBMS} + t_{gap,Q_0}, t_{Q_0,scan} + t_{Q_1,scan}) + t_{I,Q_1}. \tag{11}$$

The time $t_{I,Q_0,DBMS}$ for the DBMS operation execution can be derived from the processing rate r_{dbms} of the DBMS, which has been measured on the host.

Strategies **II** and **III** try to hide the reconfiguration time while avoiding the transfer of unnecessary data. The execution time t_{II} for strategy **II** is calculated as:

$$t_{II} = \max(\max(t_r, t_{Q_0,scan}) + t_{II,Q_0,acc_0} + t_r + t_{II,Q_0,acc_1} + \max(t_{II,Q_0,trans} + t_{gap,Q_0}, t_r), t_{Q_0,scan} + t_{Q_1,scan}) + t_{II,Q_1} \tag{12}$$

with $t_{II,Q_1} = t_{I,Q_1}$. In case of strategy **III**, we obtain:

$$t_{III} = \max(\max(t_r, t_{Q_0,scan}) + t_{III,Q_0,acc_1} + t_r + t_{III,Q_0,acc_0} + t_{III,Q_0,trans} + t_{gap,Q_0}, t_{Q_0,scan} + t_{Q_1,scan}) + t_{III,Q_1} \tag{13}$$

with $t_{III,Q_1} = t_{I,Q_1}$.

To validate our evaluation model, we have used a test setup. We have run 3 queries (Q_0, Q_1, Q_2) on our FPGA-based prototype. Q_0 has a `WHERE` clause using 2 columns and a selectivity factor of 0.1. Q_1 has a `WHERE` clause using only 1 column and also a selectivity factor of 0.1. Q_2 has a `WHERE` clause using also only 1 column, but a selectivity factor of 0.2. Our prototype has a measured accelerator rate r_{acc} of ≈1,544 MB/s. The results are shown in Table 1.

As can be seen, the number of columns used in the `WHERE` clause and the selectivity factor have no impact on the execution time of the particular accelerator. Only the size of the table the accelerator is applied to makes a difference. The subsequent operation works on the reduced dataset of the tuples fulfilling the `WHERE`clause. Its

Table 1 Comparison of measured and modeled accelerator times

table size (s) [bytes]	Query	f	t_{acc} [ms] measured	model (s/r_{acc})
1,200,000	Q_0	0.1	0.781	0.777
	Q_1	0.1	0.781	0.777
	Q_2	0.2	0.782	0.777
2,400,000	Q_0	0.1	1.542	1.554
	Q_1	0.1	1.537	1.554
	Q_2	0.2	1.537	1.554

size and thus the workload of the subsequent operation are determined by the selectivity of the previous operation.

This evaluation model is actually very close to what is really happening in our system. Our hardware accelerators are built to process the incoming data at line-rate, and each accelerator must process all incoming data. This means that processing a dataset of size $s_{Q_i,table}$ takes the time given in Eqn. 2. Moreover, the cost of predicate evaluation by the accelerator is constant in terms of time or throughput, as we have explained in Section 2. So, all in all, our proposed evaluation model is actually the basis for cost-based plan enumerations—which are however not in the scope of this paper, but of our immediate future work.

7 Evaluation

In the following, we evaluate the optimizations using the presented model and varying the variables t_{gap,Q_0}, $s_{Q_0,table}$, f_{Q_0,acc_0}, and f_{Q_0,acc_1}. Table 2 gives the ranges used for each mentioned variable and the constants used to parameterize our model. A query selectivity factor f_{Q_0} is used together with a selectivity *ratio* to determine f_{Q_0,acc_0} and f_{Q_0,acc_1} as follows:

$$f_{Q_0,acc_1} = \sqrt{f_{Q_0}} + (1 - \sqrt{f_{Q_0}}) \times (1 - ratio) \quad (14)$$

$$f_{Q_0,acc_0} = \frac{f_{Q_0}}{f_{Q_0,acc_1}} \quad (15)$$

The *ratio* determines whether the selectivity f_{Q_0} is split equally (*ratio* = 1) or is completely handled by the first accelerator (*ratio* = 0). This allows to control the relation of the two selectivities of the first query with one value. Please note that the case of $f_{Q_0,acc_1} < f_{Q_0,acc_0}$ is excluded from the evaluation, because then the query optimizer would have swapped the accelerators in the standard plan **S**. The performance improvements of the suggested optimizations of **S** have been calculated using the formulas given above and the variable values and constants of Table 2.

Table 2 Range of values explored for each variable and constants used to determine t_S, t_I, t_{II}, and t_{III}

Variable	Values	Constant	Value
t_{gap,Q_0}	0–15 ms	t_r	15 ms
f_{Q_0}	0–0.25	r_{scan}	8 GB/s
$s_{Q_0,table}$	120 KB–120 GB	r_{acc}	12 GB/s
$s_{Q_1,table}$	120 KB–120 GB	$r_{network}$	4 GB/s
ratio	0–1	r_{dbms}	10 GB/s

First, we present the achievable improvement in relation to the standard plan **S** when varying the size $s_{Q_0,table}$. The possible gains can be seen in Fig. 3. For each combination of the two table sizes, the best possible solution when applying either strategy **I**, **II**, or **III** is shown, as $\frac{t_S - \min\{t_I, t_{II}, t_{III}\}}{t_S}$. These results quantify the expected behavior of a declining improvement with increasing table sizes. There, the time spent in scanning and processing is also increasing, and the time needed for reconfiguration becomes more and more negligible. However, the benefits of speculatively starting the scan operation of Q_1 directly after the scan operation of Q_0 can yield additional improvements. A higher reduction is achieved as the table size $s_{Q_1,table}$ of the second query increases. The limit for this is reached when $t_{Q_1,scan}$ cannot be completed until Q_1 arrives. This case can be observed for small values of $s_{Q_0,table}$ and $s_{Q_1,table}$ = 2,274 MB.

While in general these improvements look promising, we also want to know which of the optimization strategies is delivering the best improvement and in which situation. $s_{Q_1,table}$ is not relevant for this evaluation and set to 1,263 MB. Figure 4 reveals which optimization yields the largest execution-time reduction with respect to a series of variable values. The time t_{gap,Q_0} between the queries is increased from left to right (0, 2, and 4 ms) and the size $s_{Q_0,table}$ of the first table increases downwards (1.5 GB in

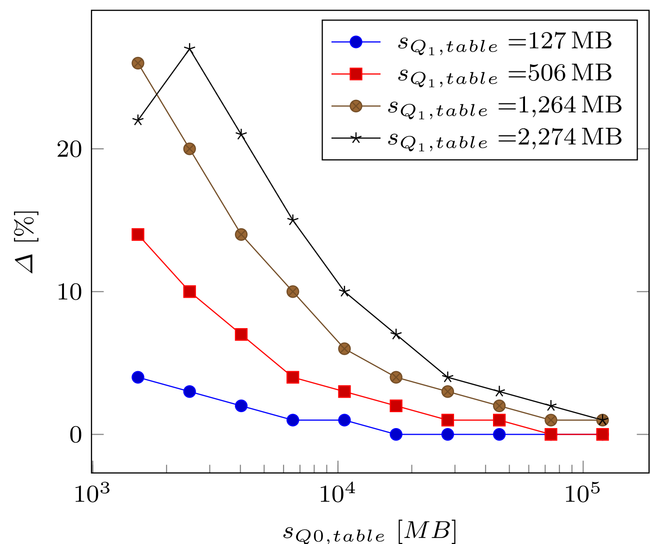

Fig. 3 Achievable execution-time improvement relative to standard plan **S** when applying the three proposed strategies, in dependence of the two table sizes. Depicted values are the best solutions found out of all strategies when varying the parameters *ratio*, t_{gap,Q_0} and f_{Q_0}

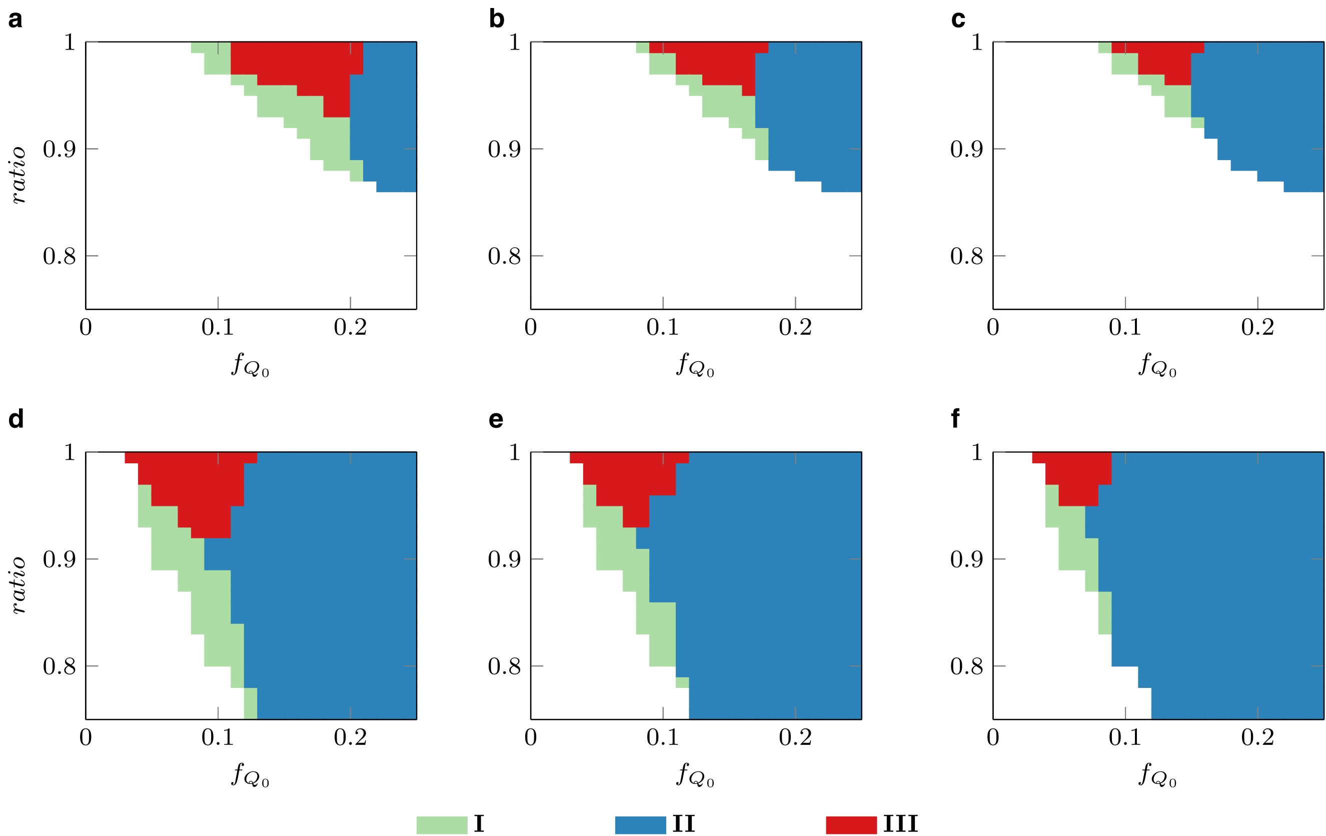

Fig. 4 Optimization strategies yielding the largest execution-time improvement in dependence of t_{gap,Q_0}, $s_{Q_0,table}$, f_{Q_0}, and *ratio*. **a** t_{gap,Q_0} = 0 ms, $s_{Q_0,table}$ = 1.5 GB. **b** t_{gap,Q_0} = 2 ms, $s_{Q_0,table}$ = 1.5 GB. **c** t_{gap,Q_0} = 4 ms, $s_{Q_0,table}$ = 1.5 GB. **d** t_{gap,Q_0} = 0 ms, $s_{Q_0,table}$ = 2.4 GB. **e** t_{gap,Q_0} = 2 ms, $s_{Q_0,table}$ = 2.4 GB. **f** t_{gap,Q_0} = 4 ms, $s_{Q_0,table}$ = 2.4 GB

the first row and 2.4 GB in the second). White regions indicate situations where the DBMS would not use the accelerators as in **S** or would not push down all operations. In Fig. 4a–c, the winner always reduces the time needed in **S** by 25–30%, whereas for Fig. 4d–f, the improvement amounts to 18–20%.

The first thing to recognize is that **II** yields the highest improvement for larger tables and for more equally distributed selectivity factors (lower *ratio*). Strategy **I** benefits from small tables and very small selectivity factors f_{Q_0}. This makes sense, as the penalty of two additional reconfigurations is larger than the penalty of using the DBMS. Strategy **III** fills the gap between **I** and **II**. It is suitable only for a very narrow subset of queries, mainly in cases where the gap between the consecutive queries is small and the accelerators of Q_0 can be swapped with almost no additional cost ($f_{Q_0,acc_0} = f_{Q_0,acc_1}$).

Since we have identified cases in which the optimization strategies are beneficial, we can generate concrete optimization rules and turn the formulas into cost models, enabling the optimizer to actually use the identified optimizations to either push down all filter operations with hints or not to push down some filter operations (strategy **I**). Exploiting knowledge about query sequences in local optimization is not only promising when scheduling the reconfigurations of accelerators on RPUs. Prefetching of data has the highest potential for execution-time reduction. Moreover, it would be possible to keep result data in the memory of the RPU, if they are needed in subsequent queries for evaluating common sub-expressions and subsumptions.

8 Future Perspectives

Knowledge of query sequences can help much in designing heterogeneous database systems that include reconfigurable hardware in the form of one RPU or even a cluster of multiple RPUs, as shown in Fig. 5. Query partitioning should distribute the query processing over several nodes (host and one or even multiple RPUs) to improve utilization, performance, and scalability, but also energy consumption, by allowing parallel queries to work on data locally without having to transfer large amounts of data across the network. However, the optimization potential is determined by (i) the RPU capabilities, i.e., the set of accelerators available on each FPGA, and (ii) the data locality, i.e., the dis-

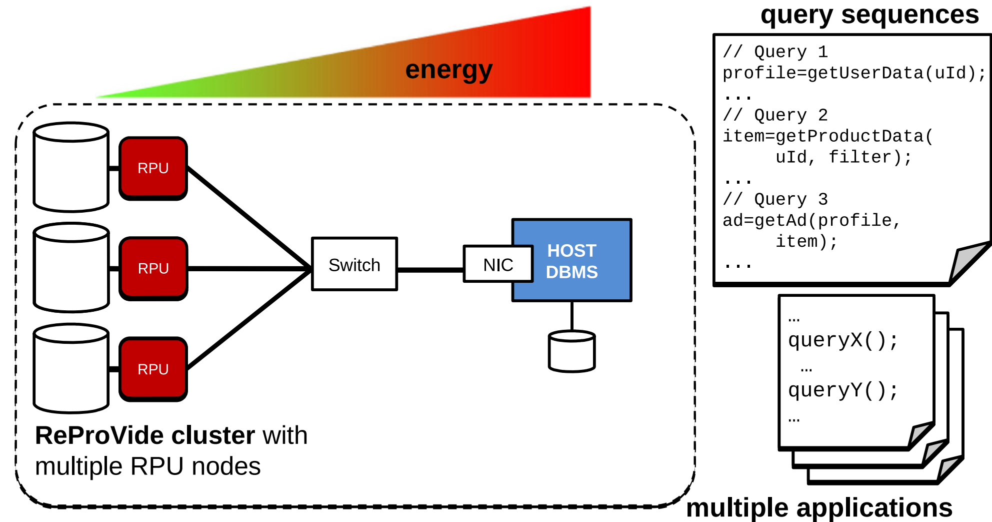

Fig. 5 Overview of a ReProVide cluster where multiple RPUs are connected to a host, which can schedule multiple applications on the cluster. Each application executes a sequence of queries

tribution of the data among the host and the RPUs. Query-sequence information helps to optimally configure the system in terms of (i) which hardware accelerators should be available on each RPU (RPU capabilities) and (ii) how the datasets should be partitioned and distributed over the system's nodes. One option is to synthesize new accelerators by combining other arithmetical and comparison operations, depending on the frequency of these combinations in the query sequences. Another option is to assign queries with strong commonalities to the same node, so that they can reuse accelerators and cached results. For larger table sizes, the RPUs can provide an index lookup instead of a full table scan. This would reduce $t_{Q_i,scan}$ significantly. Here, the information about the data access and operations in the query sequences can be used to decide which index should be created, including join indexes and materialized views.

9 Conclusion

This paper proposes the exploitation of information on query sequences in the optimization on reconfigurable accelerator hardware. Particularly, the ReProVide Processing Unit (RPU) has been introduced as a system that can filter data on their way from storage to a DBMS host. While reconfigurable hardware can process data at line-rate, the overhead for loading the right accelerator can be annoying. In this paper, we have shown how to reduce this overhead by taking the knowledge about forthcoming queries into account. We have particularly identified optimization strategies that can be used to implement heuristics for query partitioning. We have also discussed how query-sequence information can furthermore help to design these systems in terms of synthesizing hardware accelerators, data partitioning, and index generation.

Acknowledgements This work has been supported by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of the Priority Programme SPP 2037. The authors thank the reviewers for using their precious time to review the submission and for their valuable feedback.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alonso G, Roscoe T, Cock D, Ewaida M, Kara K, Korolija D, Sidler D, Wang Z (2020) Tackling hardware/software co-design from a database perspective. In: Proc. CIDR
2. Balkesen C, Kunal N, Giannikis G, Fender P, Sundara S, Schmidt F, Wen J, Agrawal S, Raghavan A, Varadarajan V, Viswanathan A, Chandrasekaran B, Idicula S, Agarwal N, Sedlar E (2018) RAPID: in-memory analytical query processing engine with extreme performance per watt. In: Proc. SIGMOD, pp 1407–1419
3. Becher A, Ziener D, Meyer-Wegener K, Teich J (2015) A co-design approach for accelerated SQL query processing via FPGA-based data filtering. In: Proc. FPT, pp 192–195
4. Becher A, Beena Gopalakrishnan L, Broneske D, Drewes T, Gurumurthy B, Meyer-Wegener K, Pionteck T, Saake G, Teich J, Wildermann S (2018) Integration of FPGAs in database management systems: challenges and opportunities. Datenbank Spektrum 18(3):145–156

5. Becher A, Herrmann A, Wildermann S, Teich J (2019) ReProVide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. In: Meyer H et al (ed) Proc. BTW – Workshopband. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik, Bonn, pp 51–70
6. Beena Gopalakrishnan Nair L, Becher A, Meyer-Wegener K (2020) The ReProVide query-sequence optimization in a hardware-accelerated DBMS (full paper). arXiv cs.DB(2005.01511)
7. Beena Gopalakrishnan Nair L, Becher A, Meyer-Wegener K (2020) The ReProVide query-sequence optimization in a hardware-accelerated DBMS (short paper). In: Proc. DaMoN Workshop
8. Beena Gopalakrishnan Nair L, Becher A, Meyer-Wegener K, Wildermann S, Teich J (2020) SQL query processing using an integrated FPGA-based near-data accelerator in ReProVide (demo paper). In: Bonifati A, Zhou Y, Salles MAV, Böhm A, Olteanu D, Fletcher GHL, Khan A, Yang B (eds) Proc. EDBT, pp 639–642
9. Bowman IT, Salem K (2007) Semantic prefetching of correlated query sequences. In: Proc. ICDE, pp 1284–1288
10. Breß S, Beier F, Rauhe H, Schallehn E, Sattler K, Saake G (2012) Automatic selection of processing units for coprocessing in databases. In: Proc. ADBIS, pp 57–70
11. Breß S, Schallehn E, Geist I (2012) Towards optimization of hybrid CPU/GPU query plans in database systems. In: Proc. ADBIS Workshops, pp 27–35
12. Breß S, Beier F, Rauhe H, Sattler K, Schallehn E, Saake G (2013) Efficient co-processor utilization in database query processing. Inf Syst 38(8):1084–1096
13. Breß S, Funke H, Teubner J (2016) Robust query processing in co-processor-accelerated databases. In: Proc. SIGMOD, pp 1891–1906
14. Chaudhari MB, Dietrich SW (2016) Detecting common subexpressions for multiple query optimization over loosely-coupled heterogeneous data sources. Distrib Parallel Databases 34(2):119–143
15. Chen FF, Dunham MH (1998) Common subexpression processing in multiple-query processing. IEEE Trans Knowl Data Eng 10(3):493–499
16. István Z, Sidler D, Alonso G (2017) Caribou: intelligent distributed storage. PVLDB 10(11):1202–1213
17. Khoussainova N, Balazinska M, Gatterbauer W, Kwon Y, Suciu D (2009) A case for a collaborative query management system. In: Proc. CIDR, p 94
18. Kraft T, Schwarz H, Rantzau R, Mitschang B (2003) Coarse-grained optimization: techniques for rewriting SQL statement sequences. In: Proc. VLDB, pp 488–499
19. Nagy C (2013) Static analysis of data-intensive applications. In: Proc. CSMR, pp 435–438
20. Najafi M, Sadoghi M, Jacobsen HA (2013) Flexible query processor on FPGAs. PVLDB 6(12):1310–1313
21. Owaida M, Alonso G, Fogliarini L, Hock-Koon A, Melet P-E (2019) Lowering the latency of data processing pipelines through FPGA based hardware acceleration. PVLDB 13(1):71–85
22. Schwab P, Wahl AM, Lenz R, Meyer-Wegener K (2016) Query-driven data integration (short paper). In: Proc. LWDA, pp 206–211
23. Schwarz H, Wagner R, Mitschang B (2001) Improving the processing of decision support queries: the case for a DSS optimizer. In: Proc. IDEAS, pp 177–186
24. Sellis TK (1988) Multiple-query optimization. ACM Trans Database Syst 13(1):23–52
25. Sidler D, István Z, Owaida M, Alonso G (2017) Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In: Proc. SIGMOD, pp 403–415
26. Smith Z (2011) Development of tools to manage embedded SQL. In: Proc. 49th ACM Annual Southeast Regional Conf. Kennesaw, GA, USA, March 24-26 ACM, New York, pp 358–359
27. Sukhwani B, Thoennes M, Min H, Dube P, Brezzo B, Asaad SW, Dillenberger D (2013) Large payload streaming database sort and projection on FPGAs. In: Proc. SBAC-PAD, pp 25–32
28. Teubner J (2017) FPGAs for data processing: current state. it Inf Technol 59(3):125
29. Vogelsgesang A, Mühlbauer T, Leis V, Neumann T, Kemper A (2019) Domain query optimization: adapting the general-purpose database system Hyper for Tableau workloads. In: Proc. BTW, pp 313–333
30. Wahl AM, Endler G, Schwab P, Herbst S, Rith J, Lenz R (2018) Crossing an OCEAN of queries: analyzing SQL query logs with OCEANLog. In: Proc. SSDBM, pp 30:1–30:4
31. Watanabe S, Fujimoto K, Saeki Y, Fujikawa Y, Yoshino H (2019) Column-oriented database acceleration using FPGAs. In: Proc. ICDE, pp 686–697
32. Woods L, István Z, Alonso G (2014) Ibex— an intelligent storage engine with support for advanced SQL off-loading. PVLDB 7(11):963–974 (proc. 40th Int. Conf. on VLDB (Hangzhou, China, Sept. 1–5))
33. Yan J, Jin Q, Jain S, Viglas SD, Lee A (2018) Snowtrail: testing with production queries on a cloud database. In: Proc. DBTest, pp 4:1–4:6
34. Yao Q, An A, Huang X (2005) Finding and analyzing database user sessions. In: Proc. DASFAA. Springer, Berlin, Heidelberg, pp 851–862
35. Ziener D, Weber H, Vogt JS, Schürfeld U, Meyer-Wegener K, Teich J, Dennl C, Becher A, Bauer F (2016) FPGA-based dynamically reconfigurable SQL query processing. ACM Trans Reconfigurable Technol Syst 9:25:1–25:24