COPRAO: A Capability Aware Query Optimizer for Reconfigurable Near Data Processors

Lekshmi B. G., Klaus Meyer-Wegener
Data Management (CS6), Friedrich-Alexander-Universität
Erlangen, Germany
{lekshmi.bg.nair, klaus.meyer-wegener}@fau.de

***Abstract*—Placing the processing power near the data, rather than shipping the data to the processor is inevitable and demanding in the era of Big Data. Thereby near-data processors gained much attention in recent years as they can reduce the massive data transfer between data sources and computing nodes. However, it is important to rethink the computing architecture to achieve the maximum advantage of near-data-processing technology, particularly for query-processing applications. In this paper, we propose a new approach to the query optimization in a relational DBMS, which considers the specialized computing capabilities of the attached FPGA-based near-data processors. The paper focuses on the hardware-conscious optimization using extended rules and cost models as well as on refining the optimization strategies for changes in the hardware state of execution. Our evaluations demonstrate that the proposed query-optimization approach can improve the processing of queries using near-data processors based on FPGAs.**

***Index Terms*—Query optimization, FPGA, Near-data processor, Hardware capability.**

I. Introduction

In this era of Big-Data applications, it is often challenging yet demanding to provide quick response to user queries from large collections of data in various sizes and types. The key to addressing this challenge is to get the best performance from the back-end databases of these applications by speeding up the time of query execution in orders of magnitude. The query optimizer, however, must perform well in order for the database systems to achieve reasonable efficiency. But that involves a great deal of hand-tuning for particular workloads and data sets, with regards to the progress made over the past decades. Furthermore, a query optimizer needs to be tediously maintained, specifically when the system's storage and execution engines evolve. Therefore, optimizer heuristics for selecting the best query-execution plan (QEP) determine the performance of a database system, while the underlying hardware that executes the query has varying capabilities according to the latest technologies. In this paper, we propose a new approach to query optimization particularly for a reconfigurable FPGA-based near-data processor, which has its own processing capabilities and data storage. Here, our main attempt is to expand the optimizer with a set of optimization rules that consider the different capabilities and states of a near-data processor connected to it.

This work has been supported by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) as a part of SPP 2037 with the grant no. ME 943/9-1.

Near-data processors allow operations to be performed mainly for avoiding massive data transfer between data sources and computing nodes, which impedes performance, scalability, and energy efficiency. ReProVide (for "Reconfigurable data ProVider") is such a near-data processor with an FPGA-based storage-attached hardware technology (the "ReProVide Processing Unit" aka. RPU), employed in synergy with a relational DBMS (RDBMS). An RPU is capable of processing queries as it has a library of operator modules, which can be configured onto the FPGA. However, only a subset is ready to process the query, and to load any other module, reconfiguration of one or more FPGA regions is required. This means that an RPU has a state with a configured set of modules, and changing that state costs additional time [1].

New global optimization methods are needed to integrate such hardware into a DBMS and to decide which operations are worth allocating to RPUs and which are not, based on their capabilities and state. For that, it is important to reconsider query-optimization strategies to include hardware features and to take the processing state into account. *Apache Calcite* [2] is an optimizer framework that comprises a typical RDBMS except data storage and that allows to extend it with own optimization rules and cost models for any number of data sources to be attached to it. Hence we have selected Calcite as our DBMS and have used it to develop an RPU-specific capability-aware optimizer, which we call *COPRAO* ("CO-PRocessor Aware Optimizer"). It identifies the hardware (co-processor) state and decides which operations are worth assigning to the hardware (query partitioning).

With the introduction of Flash technologies and reconfigurable processing elements, Smart SSDs [12] as well as FPGA-based storage engines [13], [14] target databases for efficient query processing and fast delivery of results using the concept of near-data processing [15]–[18]. However, neither of the above mentioned works have yet discussed or proposed an optimization strategy that decides query or operation offloading based on FPGA reconfiguration time or its dynamic state. While our approach has some similarity with the approach of Garlic [19], our hardware adaptations are new. Furthermore, a few more works [20], [21] considered co-processor knowledge in optimization, but their works are more directed towards GPU co-processors.

The paper is structured as follows: In Section II, we explain the capabilities of the ReProVide near-data processor. The ar-

chitecture and optimization strategies of COPRAO are detailed in Section III. Section IV illustrates and discusses evaluations of the presented concepts. Finally, Section V concludes the paper.

II. The ReProVide Near-data Processor

An RPU includes static hardware modules such as a storage controller, a network controller, local memory, data interconnects, and multiple *partially reconfigurable regions* (PRs) [3]. Data is processed by a pipeline of operator modules called *accelerators* loaded into these PRs. An RPU executes a (partial) query 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]. This implies that the cost of changing the RPU state is determined by the time it takes to reconfigure PRs to swap accelerators.

Currently, operator modules for filtering, projection, and semi-join are available for the hardware, see [3], handling integers, floats, and strings. However, some operations such as sorting or joining larger tables can be processed at greater speed by the DBMS itself. As hardware accelerators are best for line-rate processing, and FPGA resources are limited, the available operator modules cannot be combined all 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. 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., $<, >, =, \neq, \leq, \geq$.

A unique approach of the ReProVide project is that in addition to the query-execution request, the RPU interface also allows to send some *hints*. These hints do not alter any features, but give some information so that the execution can be optimized further in the DBMS as well as in the RPU. This is usually happens in idle state. In the situation when not enough PRs are available for the given query, a cost model can be used to direct the selection of operations for the RPU, as reconfiguration of a new accelerator creates additional overhead during query processing. Often, if the DBMS knows about the upcoming queries (which is the case if query sequences have been extracted from a query log), hints about the tables and attributes that will be accessed and the operations that will be required to execute these queries are sent to the RPU. This allows to load the necessary accelerators into the PRs and to fetch the required table into memory in advance, before the query arrives. So the reconfiguration overhead as well as the table-fetching time can be eliminated from the total execution time, as discussed in [4]. The working of ReProVide has been demonstrated in [5].

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

III. The CO-Processor-Aware Optimizer

A. Architecture and Overview

The architecture of COPRAO contains an RPU capability catalog, a query partitioner, a plan generator, a cost calculator, a collector, and a capability monitor, in addition to Calcite's Volcano optimizer and execution engine, as shown in Fig. 1. COPRAO operates in three phases, namely Registration, Query processing, and Adaptation.

In the *Registration phase*, the *RPU capability catalog* initializes the capabilities of the attached hardware (the RPU), which includes the operations supported, the maximum number of comparators, the list of accelerators, and the number of partial regions that are available in the RPU. These are provided by the RPU through hints when it first connects with the DBMS.

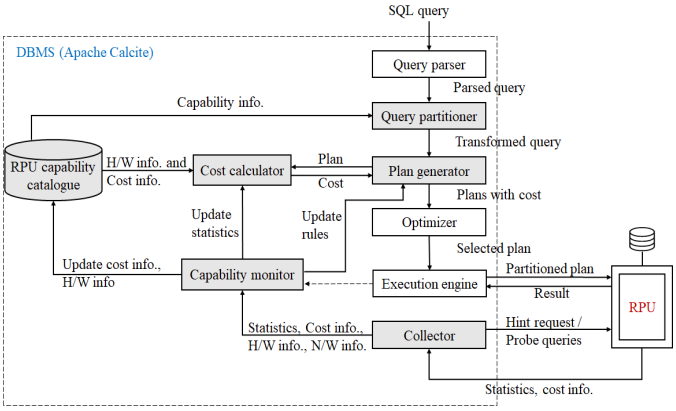


Figure 1: Overview of ReProVide

In the *Query-processing phase*, the SQL query received by the DBMS is first parsed and validated by Calcite's *Query parser*. The parsed query is then received by COPRAO's *Query partitioner*, which partitions it into fragments that can be executed either by the RPU or by the DBMS, based on their resp. processing capabilities. The transformed query is then passed to the *Plan generator*, which identifies all feasible plans based on the current hardware state. It continuously communicates with the *Cost calculator* to calculate the approximate costs of the plans. The Query partitioner and the Plan generator require a set of rules to find suitable operations in the QEP that can be worth to assign to the RPU based on the information from the catalog. Calcite's Volcano *Optimizer* then selects the best plan based on the cost and provides it to the *Execution engine*. The chosen QEP contains some operations to be executed by the RPU and others to be executed by the DBMS [5].

The *Adaptation phase* occurs after the execution (in parallel to the data transfer), where the *Collector* is responsible for collecting adequate information from the RPU. Usually, it sends a request for hints, mainly for knowing the actual cost of executing the pushed-down operations and for receiving (updated) column statistics. The RPU can generate these statistics without any additional cost [6]. The hints received as responses to the request are provided to the *Capability monitor*

to fine-tune the cost models and thus allow a better choice of QEPs in the future. In addition to this, the Collector holds a list of *Probing queries*. These are not the usual SQL statements, but lists of attributes and operations on them in a format that can be readily sent to the RPU (without the need to optimize them further). They are executed for specific purposes, such as knowing the state of the hardware or the network speed, or retrieving certain data for checking dependencies of attributes, etc. It is the responsibility of the *Capability monitor* to trigger the Collector to send the appropriate probing query and to collect its results from the Execution engine. The Capability monitor evaluates the hardware performance, updates the H/W info. and column statistics, and enables/disables rules for plan generation according to the information received from the Collector.

Furthermore, the order dependencies between the most commonly used attributes and the unique key in the resp. tables can also be determined and stored in the Capability monitor. For that, it maintains a *Data registry*, which stores the attributes, their most often accessed (range of) values as partitions (zones) and the maximum and minimum values of the unique key in each partition. It also enables the optimizer to consider additional query plans that process `Join`, `Order-by`, `Group-by`, `Distinct` operators more efficiently, as explained in [7]. We are at an initial stage of exploring efficient algorithms to determine order dependencies between attributes. What we have will be discussed below in Subsection III-C in the context of the *PredicateInduceOnJoin Rule*.

B. Cost Estimation

COPRAO chooses the best QEP based on the time required to execute both the RPU and the DBMS operations, like many other cost-driven optimizers. To determine whether or not it is worth assigning these operations to the RPU, the execution time of specific RPU operations is compared to the respective execution time of the DBMS. To determine the result cardinality of each pushed-down operation with the method explained in [8], the *Cost calculator* uses the RPU-specific cost factors recorded in the *RPU capability catalog* and the column statistics stored in the *Cost calculator* itself. According to the initial cost model of COPRAO, the total execution time of a query, $totalTime_Q$, is the sum of the execution time at the RPU, t_{RPU}, the time of the data transfer through the network, $t_{network}$, and the DBMS execution time, t_{DBMS}, for the remaining (not pushed-down) operations in the query. I. e.

$$totalTime_Q = t_{RPU} + t_{network} + t_{DBMS} \tag{1}$$

The time for executing an operation in the RPU, t_{RPU}, is basically determined by the RPU's table-scanning time, t_{scan}, the time for reconfiguring the required hardware modules, t_r, and the accelerator time (the time for data processing), t_{acc}. I. e.

$$t_{RPU} = max(t_{scan}, t_r) + t_{acc} \tag{2}$$

Only the maximum of t_{scan} and t_r contributes to the total execution time, as the table scan can be done while the required accelerator is being loaded. The values of t_{scan}, t_{acc}, and $t_{network}$ are calculated using RPU-specific cost factors, such as the table-scanning rate r_{scan}, the accelerator processing rate r_{acc}, and the rate of the data transfer over the network, $r_{network}$.

$$t_{scan} = \frac{s_{table}}{r_{scan}}\ ,\ t_{acc} = \frac{s_{table}}{r_{acc}} \text{ and } t_{network} = \frac{f \times s_{table}}{r_{network}} \tag{3}$$

Here s_{table} is the size of the table and f is the selectivity of the operation which is pushed down to the RPU. Furthermore, the RPU execution time also depends on other hardware capabilities such as the number of PRs. For example, consider the scenario where the predicates that can be pushed down to the RPU demand two accelerators, acc_0 and acc_1, for processing the data. If the #PRs available is equal to the #accelerators required (here, if 2 PRs are free), then the total time required for the RPU to execute the operations can be calculated as

$$t_{RPU} = max(max(t_{scan}, t_{r,acc_0}) + t_{acc_0}, t_{r,acc_1}) + t_{acc_1} \tag{4}$$

The first accelerator reconfiguration can be done in parallel to the table scan and the second reconfiguration can be done in parallel to these two plus the first accelerator run. Hence, the second t_r can be hidden to a large extent. Here t_{r,acc_i} is the reconfiguration time for acc_i, $i \in \{0, 1\}$. But if the #PRs available is less than the #accelerators required (i. e. only 1 PR is free), then the total RPU time is

$$t_{RPU} = max(t_{scan}, t_{r,acc_0}) + t_{acc_0} + t_{r,acc_1} + t_{acc_1} \tag{5}$$

So in this scenario an overhead of one additional t_r is included in the total RPU processing time.

COPRAO examines the hardware state (primarily using the history of the last operations conducted in the RPU) each time a query is optimized for the RPU. This includes the accelerators currently loaded in the PRs, the availability of PRs, etc. Based on this it decides which cost model would be accurate to predict the cost of the operations to be pushed down to the RPU now. If the RPU has been provided with hints about the upcoming queries such as tables and attributes that may be requested, and operations that may be invoked, the RPU can prefetch the required tables and reconfigure or reuse the accelerators. Especially in the case of query sequences, the upcoming query can easily be predicted. The cost model again changes in this scenario, as it has already been discussed in [4].

C. Query Optimization

Query optimization in COPRAO is driven by a set of rules, which are explicitly defined based on the RPU capabilities. When a query is received by the DBMS, Calcite applies these RPU-specific rules, which are integrated into the Calcite optimizer through the adapter, in addition to its own optimization

guidelines [5]. The operations that can be pushed down to the RPU (RPU-capable expressions) are mostly identified with the help of these rules. They are mainly:

RPUOperator Rule: This rule is used in the Query partitioner to decide which operations in the QEP can be pushed down to RPU based on the information in the RPU capability catalog. For example, in the sample query[2]

```
SELECT * FROM date_dim
WHERE d_year > 1998 AND d_year < 2002
AND d_dow = 3
ORDER BY d_year,
```

the `ORDER BY` clause cannot be pushed down to RPU as it does not support the `Sort` operation, and hence that is assigned to the DBMS. So the plan will be partitioned into two, one part for the RPU-specific operation nodes and the other for the DBMS-specific nodes, as we have shown in [5].

RPUFilterOnRank Rule: This rule is applied when not enough hardware resources are available for executing a filter predicate. For instance, if the number of comparators available in the RPU is 2, then the RPU cannot execute all filter conditions of the above predicate expression, because the attributes come together in a single beat (according to the TPC-DS table schema) in the accelerator and demand 3 comparators. Here, for each potential expression in the `WHERE` clause that can be pushed down, a *rank* is determined based on the cost of executing these expressions in the RPU (t_{RPU}) and their selectivity (f_{exp}).

$$rank_{exp} = \frac{f_{exp} - 1}{t_{RPU}} \tag{6}$$

If the predicate selectivity is smaller, then more tuples would be filtered out. Likewise, if the predicate is cheaper, the benefits can be gained at low costs. Hence the expression with the lowest rank is chosen to be pushed down. Such kind of ordering was found to be efficient in [9].

RPUSpoolFilter Rule: This rule is applied when multiple accesses to the same table have been found especially in a self join. Calcite encourages to create a `Spool` node then, which prevents multiple operator push-downs for the same table by allowing intermediate results to be stored. This rule uses the facility to create a `Spool` node that takes all RPU-capable filter expressions of the same table and applies the `OR` clause to merge them, taking into account common sub-expressions and subsumptions. The RPUFilterOnRank Rule is also applied here to determine the suitable expressions for pushing down to the RPU, if not enough hardware modules are available for all the expressions in the `Spool` node.

PredicateInduceOnJoin Rule: This rule is applied to multi-joins, if some of the tables involved are not filtered. For example, consider the following query over the TPC-DS schema:

```
SELECT * FROM date_dim, store_sales
WHERE d_date_sk = ss_sold_date_sk
AND d_year = 2000;
```

Here `date_dim` is a dimension table with the primary key `d_date_sk`, and `store_sales` is a large fact table with `ss_sold_date_sk` as a foreign key referencing `date_dim`. The table `store_sales` is not filtered in this example. Since the RPU does not support the `Join` operator, the whole table would be transferred to the DBMS, which is not efficient. To avoid this, a new predicate is created for filtering `store_sales` using its join attribute `ss_sold_date_sk`. It relies on a dependency between the join attribute `d_date_sk` of the filtered table and its selection attribute `d_year`. If this dependency exists and is stored in COPRAO's Data registry, then for the range of values of `d_date_sk` $[r_1..r_2]$ in the partition of tuples that satisfy `d_year = 2000`, the rule creates a new predicate on `ss_sold_date_sk`, namely `ss_sold_date_sk BETWEEN` r_1 `AND` r_2.

TransformJoinIntoFilter Rule: In some cases, a `Join` query demands only few attributes from the large table:

```
SELECT SUM(ss_ext_sales_price)
FROM date_dim, store_sales
WHERE ss_sold_date_sk = d_date_sk
AND d_year = 2000;
```

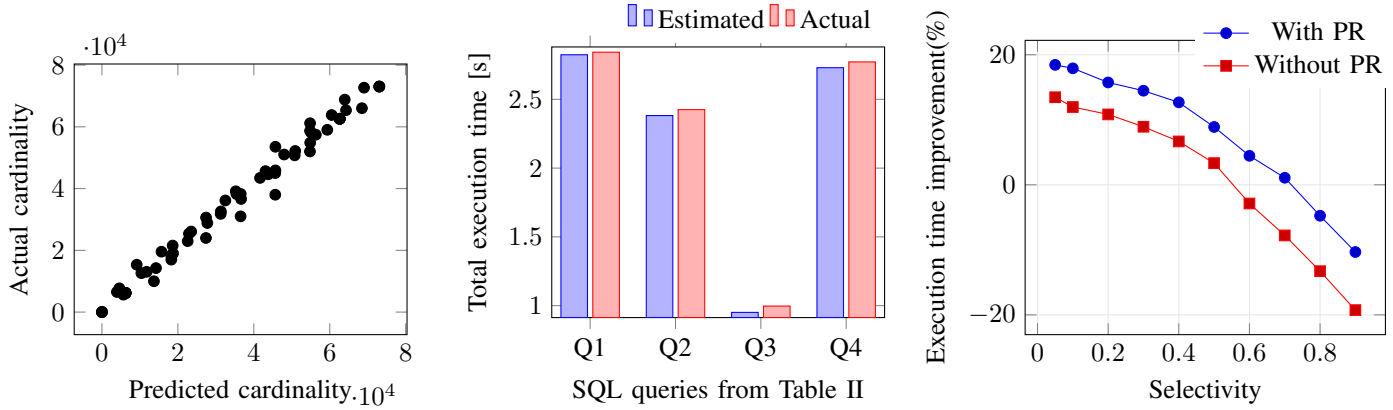
This rule can eliminate the expensive `Join` operation as well as the filtering on `date_dim` from the QEP by adding a new filter predicate, as mentioned above, according to the order dependency in the Data registry. This was found to be an effective optimization in [10].

InduceSemijoinBeforeJoin Rule: If no attribute dependency exists in the Data registry, COPRAO uses this rule in a multi-join operation to induce a semi-join in the QEP, which can be pushed down to the RPU.

D. Hardware-adaptive Plan Optimization

In its current state, COPRAO adjusts the optimization strategies primarily in two situations. In the first scenario the Capability monitor identifies that the calculated RPU cost, t_{RPU}, is different from the actual RPU cost, which is received as a hint after executing a query. If there is a change in the RPU-specific cost factors, it updates them in the catalog using the values provided in the hints. The difference in cost can also be due to the lack of enough PRs, because of the hardware workload (which may change), and this may increase the cost by additional reconfiguration time, as in (5). The Capability monitor requests hints to confirm the unavailability of PRs, and if that is the case, it applies an additional rule called the **RPUFilterOnReconfiguration Rule** to the optimization of the next query. In this rule, only those expressions for which the PRs are already loaded with the required accelerators are considered for pushing down, saving the time of reconfiguration. As a result, the cost of pushed-down operations changes to that of (2). The Capability monitor continuously evaluates the

[2]Using the schema of the well-known TPC-DS benchmark.



(a) Actual cardinality vs. Predicted cardinality (b) Estimated vs. actual total execution time as in (1) (c) Performance of InduceSemiJoinBeforeJoin Rule

Figure 2: Performance of COPRAO

hardware state to change the cost model without affecting any query processing. Multiple communication ports and transfer ports allow the RPU and the DBMS to communicate hints simultaneously.

The second scenario occurs when the Capability monitor notices that the network takes longer to transfer the data. In such a situation, COPRAO disables the RPUFilterOnReconfiguration Rule to push down all RPU-capable operations, even if this generates additional t_r, in order to reduce the size of the data transferred over the network. In addition to this, COPRAO enables a new rule called the **PredicateReplaceOnFilter Rule**. It evaluates whether an attribute used in the predicates has an order dependency with the primary key. If so, that selection can be replaced by the resp. condition on the primary key. I. e. consider the example used in description of the RPUOperator Rule. After enabling the PredicateReplaceOnFilter Rule, the `d_year` in the predicate can be replaced with the unique attribute `d_date_sk` with a condition depending on the range in which `d_year > 1998 and d_year < 2002` occurs. Then the predicate in the `WHERE` clause changes to `d_date_sk BETWEEN` r_1 `AND` r_2 `and d_dow = 3` where $[r_1..r_2]$ is the range of `d_date_sk` values in which `d_year` has the required values. Now RPU can process all the data using these predicates and the size of the data to be transmitted across the network is thereby further reduced. Usually COPRAO adapts to the hardware state offline, when no query is under processing.

IV. Evaluations

COPRAO has been implemented with Apache Calcite release 1.21.0 as a DBMS on an Intel Core i9-7920x CPU 2.90GHz × 24 processor. The first version of the RPU is connected to the DBMS using an Ethernet capable of a maximum speed of 4 GB/s. We have evaluated our rules and cost model for different queries with the values of cost factors and variables as mentioned in Table I.

First we have considered a set of queries with predicates in the `WHERE` clause as shown Table II, varying the constant

Table I: Range of values explored for each variable and constants used in cost models

Variable	Values	Constant	Value
$r_{network}$	100 MB/s – 4 GB/s	t_r	25 ms
f	0.05 – 0.9	r_{scan}	8 GB/s
s_{table}	1 GB – 10 GB	r_{acc}	12 GB/s

values in the comparisons for different selectivities ranging from 5–90%. The estimated result cardinality (#rows), which is calculated using the column statistics and the methods explained in [8], has been evaluated against the actual result cardinality. This is shown in Fig. 2a. Here we can see a very high correlation between the predicted cardinality and the actual cardinality. This is because of the accurate and regularly updated statistics of columns received from the RPU.

Table II: Predicates in the `WHERE` clause

Query	Predicate
Q1	`d_month_seq < 2000 AND d_week_seq > 1000`
Q2	`d_moy = 5 OR d_dow > 2`
Q3	`d_year > 2000 AND (d_qoy > 3 OR d_dow > 4)`
Q4	`d_year > 2000 OR (d_qoy < 3 AND d_dow <> 4)`

For the table sizes given in Table I and the filter predicates listed in Table II, we have assessed the total execution time of the queries. Fig. 2b illustrates the execution-time comparison for a table size of 10 GB. We can see here again that the time can be accurately calculated by our initial cost models. This is because the result cardinality is determined properly from the hints, and the variables used in calculating the costs have been adjusted appropriately. The time spent in packing and unpacking data requests and data packets is responsible for the differences in Fig. 2b. Fig. 2c shows the percentage of improvement when applying the InduceSemiJoinBeforeJoin Rule to the `Join` query provided in the description of the PredicateInduceOnJoin Rule. The evaluation shows that, if the selectivity is very small, inducing a semi-join before the join yields around 18% execution-time improvement, if enough

PRs are available on the RPU, and around 14% improvement with additional t_r overhead. These benefits become smaller as the size of the result increases, due to the network overhead for result transfer, the DBMS overhead for the additional `Join` operation, and the t_r overhead if not enough PRs are available.

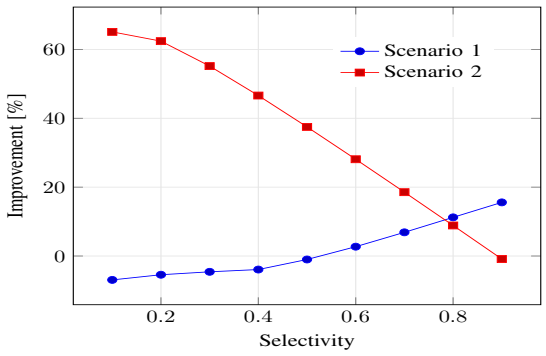


Figure 3: Performance improvement in adaptive optimization

We have modeled the Scenarios 1 and 2 to test the efficiency improvement with the rules applied in the adaptive process and for different table sizes. The results are plotted in Fig. 3. We have considered queries that require two accelerators for execution in the RPU. The efficiency of the RPUFilterOnReconfiguration Rule is evaluated by changing the total selectivity from 5% to 90% and using the fast 4 GB/s network. The assessment reveals that a maximum enhancement of 16% is reached for selectivities above 70% and for a table size of 10 GB. For smaller result sizes, pushing down only one operation does not bring benefits because of the DBMS and network overhead. Hence for small selectivities pushing down both operations is beneficial even with an additional t_r overhead.

For Scenario 2, we have considered the same state of the hardware at a reduced network speed up to 100 MB/s and have evaluated the push-down of all operations, involving additional reconfiguration by disabling the RPUFilterOnReconfiguration Rule and enabling the PredicateReplaceOnFilter Rule. This yields no gain for selectivities above 80%, since a tremendous amount of data must be transmitted at a slower network pace. Yet we see advantages for a selectivity up to 80% and can achieve around 64% in execution-time improvement for a selectivity range below 20% and for a table size of 10 GB. This decreases considerably as the selectivity increases and thus more data must be transmitted over a weak network pace. So all in all the RPU can be utilized better with the rules and cost models of COPRAO, even for a poor network-transmission rate and limited hardware resources.

V. Conclusions

Optimizing queries for new hardware has been an emerging topic of discussion for a few years. In this paper, we propose a new optimization concept, COPRAO, that can be incorporated with an RDBMS to integrate an FPGA-based near-data processor for fast and energy-efficient processing of massive data. COPRAO continuously interacts with the hardware to know and update its capabilities for efficient optimization. The advantages of this approach are extensibility and a potential of evolution for upcoming versions of the hardware. In the future, we will extend COPRAO to integrate a cluster of RPUs with different capabilities and to process streaming data in addition to stored data.

References

[1] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, "Relational query processing on OpenCL-based FPGAs," in *Proc. FPL*, 2016, pp. 1–10.

[2] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proc. SIGMOD*, 2018, pp. 221–230.

[3] A. Becher, A. Herrmann, S. Wildermann, and J. Teich, "ReProVide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing," in *Proc. NoDMC*, 2019, pp. 51–70.

[4] L. Beena Gopalakrishnan Nair, A. Becher, S. Wildermann, K. Meyer-Wegener, and J. Teich, "Speculative dynamic reconfiguration and table prefetching using query look-ahead in the ReProVide near-data-processing system," *Datenbank-Spektrum*, vol. 21, 2021, accepted for publication.

[5] L. Beena Gopalakrishnan Nair, A. Becher, K. Meyer-Wegener, S. Wildermann, and J. Teich, "SQL query processing using an integrated FPGA-based near-data accelerator in ReProVide (demo paper)," in *Proc. EDBT*, 2020, pp. 639–642.

[6] A. Becher and J. Teich, "In situ statistics generation within partially reconfigurable hardware accelerators for query processing," in *Proc. DaMoN*, 2019, pp. 1–3.

[7] J. Szlichta, P. Godfrey, and J. Gryz, "Fundamentals of order dependencies," *arXiv preprint arXiv:1208.0084*, 2012.

[8] T. Grust, "Cardinality estimation: How many rows does a query yield?" Online, Nov. 2011, https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws1011/db2/db2-selectivity.pdf.

[9] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in *Proc. SIGMOD*, 1993, pp. 267–276.

[10] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Effective and complete discovery of bidirectional order dependencies via set-based axioms," *The VLDB Journal*, vol. 27, no. 4, pp. 573–591, 2018.

[11] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[12] 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 *Proc. SIGMOD*, 2013, pp. 1221–1230.

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

[14] P. Francisco *et al.*, "The Netezza data appliance architecture: A platform for high performance data warehousing and analytics," 2011.

[15] O. O. Babarinsa and S. Idreos, "JAFAR: Near-data processing for databases," in *Proc. SIGMOD*, 2015, pp. 2069–2070.

[16] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proc. DaMoN*, 2015, pp. 1–10.

[17] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. Asaad, and D. Dillenberger, "A hardware/software approach for database query acceleration with FPGAs," *Int. Journal of Parallel Programming*, vol. 43, no. 6, pp. 1129–1159, 2015.

[18] R. Mueller, J. Teubner, and G. Alonso, "Glacier: a query-to-hardware compiler," in *Proc. SIGMOD*, 2010, pp. 1159–1162.

[19] V. Josifovski, P. Schwarz, L. Haas, and E. Lin, "Garlic: a new flavor of federated query processing for DB2," in *Proc. SIGMOD*, 2002, pp. 524–532.

[20] S. Breß and G. Saake, "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS," *Proc. VLDB*, vol. 6, no. 12, pp. 1398–1403, 2013.

[21] S. Zhang, J. He, B. He, and M. Lu, "Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures," *Proc. VLDB*, vol. 6, no. 12, pp. 1374–1377, 2013.