# Acceleration of SQL Restrictions and Aggregations through FPGA-based Dynamic Partial Reconfiguration

Christopher Dennl, Daniel Ziener, Jürgen Teich
*Hardware/Software Co-Design, University of Erlangen-Nuremberg, Germany*
*Email: {daniel.ziener, teich}@cs.fau.de*

***Abstract*—SQL query processing on large database systems is recognized as one of the most important emerging disciplines of computing nowadays. However, current approaches do not provide a substantial coverage of typical query operators in hardware. In this paper, we provide an important step to higher operator coverage by proposing a) full dynamic data path generation for support also complex operators such as restrictions and aggregations. b) Also, an analysis of the computation times of a real database queries when running on a normal desktop computer is proposed to show that c) speedups ranging between 4 and 50 are obtainable by providing generative support also for the important restrict and aggregate operators using FPGAs.**

## I. Introduction

One of the big challenges of the IT sector today is the handling of the ever growing amount of data. The annual amount of data which is worldwide created, captured, or replicated increases by a factor of 10 every five years [1]. Most of these data are stored in *relational database management systems* (DBMS) which are able to load, analyze, and process the data based on requested inquiries, mostly given in a form of *SQL queries* [2].

Consequently, it is worthwhile to investigate the usage of FPGAs in the field of database applications for query execution purposes. The main goal is not to execute the whole query with all different operations on a single FPGA. It is rather a *HW/SW co-design*, where in the first step stream-based operations, like *restrictions* and *aggregations*, are executed inside the FPGA in order to reduce the amount of data the CPUs have to process. In the second step, the remaining operations may be executed in software which hopefully can be done in a much more efficient waydue to a considerably reduced data size. To analyze data in huge tables, which are common for *data warehouse applications* [3], the offload of *restrictions*, *joins*, and *aggregations* to an FPGA may accelerate the execution of a query enormously, but has presently not addressed.

Moreover, a static hardware approach is not suitable to cover all operator types of streaming operations. With the help of *dynamic partial reconfiguration*, the approach of query-specific data path generation has been proposed [4]. Partial modules stemming from a hardware library of SQL operators can be composed on-the-fly to form a query data path and loaded on the FPGA. Moreover, this library can be further extended to new operations or SQL constructs. In [4], also a prototype of such an approach for restrictions on a stand-alone Virtex-II FPGA has been presented.

In this paper, we will present the investigation of different types of aggregations, combined with restrictions to a query accelerating system based on a Virtex-6 FPGA by using partial dynamic reconfiguration. Furthermore, we will present the coupling and the integration of such a system into a host PC with *PCI-Express* (PCIe) in order to allows a partitioning of SQL operations into partial hardware modules and software tasks.

## II. Related Work

First, existing work considering hardware acceleration of SQL query processing is summarized. For instance, Netezza *FAST engines* [5] is a commercially available platform using FPGA support for query acceleration but they only provide a limited-parameterizable static system which only covers projection and restriction operators up to now. Another approach is provided by *Glacier*, a SQL-to-VHDL compiler, which can compile SQL queries into synthesizable VHDL [6]. Furthermore, Takenaka et al. showed in [7] how to use FPGAs for SQL-based event processing which can be described in C++. Moreover, [8] investigates FPGA-based storage engines for database systems. However, the need of a complete synthesis makes all the above approaches only useful for environments with a fixed and known query set. Static hardware designs are inappropriate for two reasons: They either support only a small subset of query operators or, on-the-fly synthesis of query-specific designs might take far too much time. A first approach to use partial reconfiguration for the acceleration of SQL processing by generating query-specific data paths on-the-fly has been proposed in [4], but up to now, the approach covers only projection and restriction operations.

A very comfortable flow for building partial reconfigurable systems is the tool *ReCoBus-Builder* [9]. The proposed approach also relies on such a tool *GoAhead* [10], a successor of ReCoBus-Builder, supporting also newest Xilinx FPGA generations.

## III. Concept and System Overview

In the following, the overall system architecture is described. Multiple hard drives storing the database are attached to the host PC via a SATA interface whereas the FPGA is connected to the host PC via PCIe. Data transfers between host and FPGA are done using DMA support on the hosts main memory. The data base table contents which are stored on the disks are transferred to the main memory and from there streamed to the FPGA over PCIe. After the hardware query processing, the FPGA stores the (partial) results into the main memory for further processing in software. The limiting factor of this kind of architecture is the communication bottleneck between the disks, main memory, and the FPGA.

For practical use, it is of utmost importance that it is possible to integrate an SQL accelerator approach into an existing DBMS. The user inputs a query to the DBMS and the DBMS has to decide which parts of the query should be executed on the host or computed inside the FPGA (see Figure 1). This is done by the *optimizer* of the DBMS which generates the execution plan and is responsible to choose the best implementation to process the query as fast as possible. The hardware part of this execution plan is processed by the *reconfiguration manager* also running on the host. Operator modules from a module library which reside in the main memory of the host are selected and the query data path is composed on-the-fly. After that, the composed data path that may consists of up to 16 different modules are loaded into a partial area inside the FPGA. Furthermore, the reconfiguration manager keeps book about the allocation of the partial areas, active queries, and finished queries, etc. As soon as the entire query has been processed, the result is presented to the user.

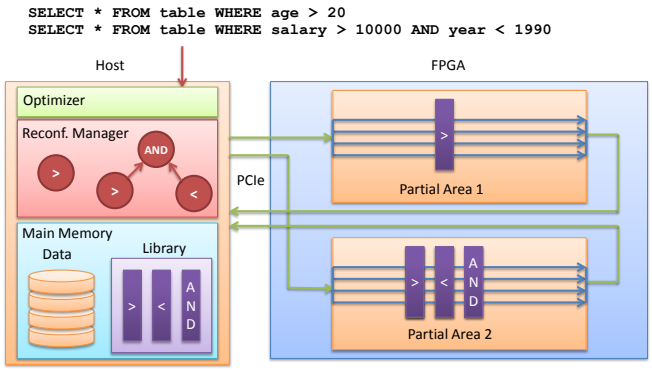


Figure 1. An overview of the proposed query acceleration system: The incoming queries are analyzed by the optimizer and reconfiguration manager and the corresponding modules are loaded. Here, only two partial areas are depicted allows the simultaneous processing of two (partial) queries in parallel.

The partial areas consist of communication structures called I/O bars. The basic principle is that each module is able to read the incoming data, modify it if necessary, and pass it further to the next module. The data is processed by the loaded operator modules and the corresponding result table is streamed continuously back to the main memory. Each partial area may implement exactly one active query accelerator, and each accelerator may consists of one or more partial modules. Figure 2 finally shows the partitioning of a partial area into so-called *slots* to deliver the mentioned stream-based communication architecture (see Section V for the slot size). One module in the operator library may consume one or more neighboring slots and multiple concatenated modules finally provide one specific query accelerator. It is also possible to configure more than one accelerator into a single area in order to hide the reconfiguration time and switch faster from one accelerator to another one. However, only one of the configured queries may be active at a time for each partial area and the modules from the inactive query are switched to bypass-mode which means that the data streams through the module without modifications.

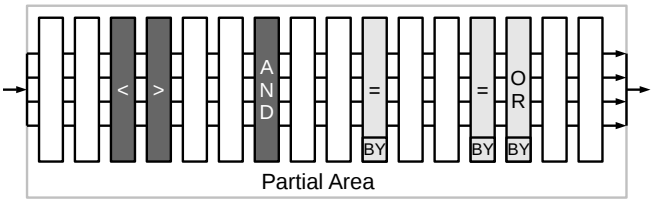


Figure 2. The slot-based partitioning of a partial area. An example module placement for processing the active query shown in Listing 1 above (dark gray). An actually still inactive, pre-loaded query in bypass mode (BY), shown in Listing 1 below, (light gray) is also depicted.

```
SELECT * FROM t WHERE x < 0 AND y > 0
SELECT * FROM t WHERE x = 0 OR y = 0
```

Listing 1. The active query (above) and the inactive one (below)

Figure 2 shows an example with one actively configured (dark gray) accelerator and one yet inactive (light grey) accelerator. The placement order of operator modules inside the partial area from left to right may be determined by the post-traversal linearization of the query expression tree (see [4] for more details).

Inside the FPGA, *tuples* are processed by the modules in a stream-based manner. Usually, the bit width of a tuple may be wider than the width of communication data bus. Therefore, tuples are divided into smaller entities called *chunks* which match the data bus width.

Currently, the library consists only of modules for processing restrictions, including arithmetics ($+$,$-$,$*$), comparators ($<$,$>$,$=$,$\neq$) and bitwise functions ($AND$, $OR$, $NOT$, $XOR$, $NAND$, $NOR$), as well as aggregations. The architectural concepts from [4] are extended to implement the restrictions and arithmetic modules. Hereby, the data path width is increased by a factor of 4 from 32 to 128 bit and by using a newer FPGA generation (Virtex-6), the clock frequency is increased from 100 MHz to 125 MHz. Moreover, the new modules for handling aggregations are developed from the scratch and will be presented in Section IV. As far as we know, this is the first approach to handle aggregations on FPGAs by using partial dynamic reconfiguration.

## IV. Module Design for Aggregations

Aggregations like **SUM**() or **MAX**() are operations to compute statistical data of multiple tuples of one or more tables (see Listing 2). In this paper, we consider the following aggregation functions because they are very common in databases: **SUM**(), **COUNT**(), **MIN**(), and **MAX**(). Optionally, aggregations can be combined with a **GROUP BY** clause to compute aggregated values for groups of data, which are separated by one or more attributes specified in the clause (see Listing 3).

```
SELECT SUM(salary) as salary_sum FROM employee
```

Listing 2. Example Query: Summation of salaries

```
SELECT department, SUM(salary) as salary_sum
FROM employee GROUP BY department
```

Listing 3. Summation of salaries grouped by department

The main difference of two cases is whether an aggregation is accompanied by grouping or not. Moreover, it

will be shown that the difference between *single-column* and *multi-column grouping* is only a minor one because technically multi-column grouping only results in a higher count of groups and the group key is only a compound key. For aggregations without grouping, only the position of the affected attribute, e.g. *salary* for the query shown in Listing 2, inside a tuple and chunk is needed. As soon as the entire table has been processed, the aggregated value also has been computed and can be appended directly to the end of the stream. If aggregations contain grouping, not only one value must be computed but one for each group. This is more complicated because it is generally not sufficient to store only one value but one for each group. In the following, a sorting-based approach is described:

1: Sort tuples by their group keys
2: $current\_group = nil$
3: **while** tuple available **do**
4:  **if** $current\_group \neq group\_key$ **then**
5:  **if** $current\_group \neq nil$ **then**
6:  Send $current\_value$
7:  **end if**
8:  $current\_value = initial(attr)$
9:  $current\_group = group\_key$
10:  **else**
11:  $current\_value = aggr(current\_value, attr)$
12:  **end if**
13: **end while**
14: Send $current\_value$

If all tuples are sorted by their respective group keys, we can exploit the fact that one group arrives one after another. Thus, as soon as one group has streamed through the data path, the aggregated value is computed and can be sent along with the group. Sorting large amounts of data inside FPGAs is a very resource intensive task. The work in [11] proposes such a sorting algorithm. However, this approach can be used anyway but data must either be sorted at run-time in software or already be stored in a sorted manner on the disk. This is the case if many queries requests data often in the same way. Then, the DBMS might reorder the storage of data to be able to answer such queries faster.

The proposed designs for computing aggregations in hardware can be combined easily with all already existing operators like restrictions and projections because they fit the stream-based manner. The restriction modules do cover arithmetic-logical operations as well as comparisons, whereas the latter one can compare both integer and string data types. Furthermore, it is not only possible to aggregate over single attributes, but also over complex arithmetic expressions. For instance, the query shown in Listing 4 could now be executed entirely in hardware.

```
SELECT MAX(x*y-z+w) as agg FROM t
WHERE x < 0 AND y+w > 0 GROUP BY z, y
```

Listing 4. Query containing restriction & aggregation & grouping operators

## V. Experimental Results

In this section experimental results are presented for the SQL accelerating approach. Performance is assured by processing different queries and compared to a software-only approach. Furthermore, two different kinds of queries are chosen to test the system. The first class are *self-made queries* which contain only operators which can be processed completely in hardware. We rely on our *employee* example from the beginning for this purpose. Second, we take a complex query from the official *TPC-DS benchmark* [12] which cannot be executed completely inside the FPGA, due to the lack of operator coverage. Therefore, we compute a *hardware/software partition* of the query and compare the execution times of a software-only run and a partially hardware accelerated one. The hardware and software setup are given as follows: Host: Core i7 920, 2.66 GHz CPU with Ubuntu 12.04 LTS with a standard MySQL 5.5 installation; FPGA: Xilinx ML605 Board con. via PCIe x4 Gen. 2

To avoid disk I/O impacts, the analysis is based on INMEMORY tables which are held in RAM. The static system consists of four partial areas, the PCIe controller, and a hardware part of the reconfiguration manager to configure the system. The hardware part of the reconfiguration manager is responsible to load the partial modules over ICAP and configure the module parameters as well as the configuration of the communication interface between the PCIe interface and the partial areas. Four partial areas can be configured to process four queries at the same time. Alternatively, the partial areas may be concatenated in order to cover queries with higher processing depth.

Finally, each partial area is divided into 16 slots with each slot having a height of 40 CLBs and a width of 2 CLBs or 1 CLB + 1 DSP/BRAM unit. Thus, one slot contains 640 or 1280 flip flops and 320 or 640 LUTs. The restriction and aggregation modules without grouping need one slot, whereas the aggregation modules with grouping need two slots. Furthermore, each partial area has a 128 bit data bus. The system operates at 125 MHz which gives a total throughput of 2 GB/s per partial area. By utilizing all four partial areas with independent queries, therefore a theoretical maximum throughput of 8 GB/s is obtained. The PCI-Express bus is able to deliver a netto transfer rate of 1.7 GB/s for each direction, i.e., the bus is clearly the bottleneck in this test setup.

```
SELECT department, years_in_company, SUM(salary) as sal_sum
FROM employee GROUP BY department, years_in_company
```

Listing 5. Example query summing up salaries grouped by department and seniority

```
SELECT department, years_in_company, SUM(salary) as sal_sum
FROM employee WHERE years_in_company BETWEEN 10 AND 30
GROUP BY department, years_in_company
```

Listing 6. Query containing a restriction & aggregation with grouping

The synthetic queries are shown in Listing 2, 3, 5, and 6. The table $employee$ is defined as $(id, salary, department, years\_in\_company)$. Each attribute is a 32 bit integer, thus one employee tuple has a size of 16 byte. The table contains about 16 million employees which gives a total size of 256 MB for this table.

Finally, query 52 from the TPC-DS benchmark [12] shown in Listing 7 is taken because it contains common operators which occur almost always in any business analyt-

ics SQL query, i.e., join, restriction, aggregation, grouping, projection, and sorting.

```
SELECT dt.d_year, item.i_brand_id brand_id, item.i_brand brand,
  SUM(ss_ext_sales_price) ext_price
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk AND
  store_sales.ss_item_sk = item.i_item_sk AND
  item.i_manager_id = 1 AND dt.d_moy = 11 AND dt.d_year = 2000
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
ORDER BY dt.d_year, ext_price DESC, brand_id
```

Listing 7. Query 52 from TPC-DS benchmark [12]

The entire data definition and schemes of the TPC-DS benchmark are described in [12]. Data is generated using the tool *dsdgen* which is shipped with the benchmark. We generate a 1 GB data set for the benchmark database.

First, the *non-benchmark queries* are investigated. The query is executed in software and hardware. For both cases, we estimate the time until the result is available in the host memory, i.e., in case for the FPGA the complete round-trip from host to FPGA over PCIe and back. Table I shows the obtained speedups. As can be seen, grouping has quite an impact on the run-time in software, whereas it has no impact in hardware because all modules have the same throughput and the entire stream processing is pipelined. In fact, only the PCIe bus limits the execution time because it may not fully utilize the maximum bandwidth of the partial modules.

Table I
EXECUTION TIMES FOR NON-BENCHMARK QUERIES

| Query | Software | Hardware | Speed-up |
|---|---|---|---|
| Listing 2 | 2.6s | 0.22s | **x12** |
| Listing 3 | 9.53s | 0.22s | **x43** |
| Listing 5 | 11.8s | 0.22s | **x54** |
| Listing 6 | 8.28s | 0.22s | **x38** |

The analysis of the *TPC-DS benchmark query* is a bit more complicated because we cannot run it completely in hardware. Rather, only the restrictions can be computed because the aggregation must be computed after joining the tables. To measure the *hardware/software case*, we create *views* for the software which represent the partial result which would be delivered from the hardware by executing parts of the query inside the FPGA. Afterwards, a modified query is executed which does not contain the parts already executed in hardware. Listing 8 and 9 show how to create the views.

```
CREATE VIEW vi AS
SELECT * FROM item WHERE item.i_manager_id=1
```

Listing 8. View for item table

```
CREATE VIEW vd AS SELECT * FROM data_dim dt
WHERE dt.d_moy = 11 AND dt.d_year = 2000
```

Listing 9. View for data_dim table

Table II shows the different estimated execution times. The software-only solution executes the unmodified query shown in Listing 7. In the mixed solution, the hardware creates the results of the views and the software executes the rest. Although the query cannot be executed entirely in hardware, it benefits from the partial hardware acceleration.

Table II
EXECUTION TIME FOR THE TPC-DS BENCHMARK QUERY

| Query / View | Software | Hardware | Speed-up |
|---|---|---|---|
| Listing 7 | 0.78s | - | - |
| Views (Listing 8 and 9) | - | 0.01s | - |
| Software part of Listing 7 | 0.15s | - | - |
| Combined | 0.16s | | x4.8 |

In comparison to the Virtex II Pro implementation presented in [4] which provides a single partial area with 400 MB/s throughput, our system supports queries placed in multiple partial areas each obtaining a throughput of 2 GB/s.

## VI. CONCLUSIONS & FUTURE WORK

In this paper, we presented concepts for on-the-fly hardware acceleration for the important class of restrict and aggregate operators appearing in many SQL queries. Using partial dynamic reconfiguration, query-specific data paths are composed on the fly. Impressive speedups of up to 50 have been reported on real database queries showing that hardware acceleration becomes possible for the first time for a substantial amount of query operators. In the future, we are trying to take the last leap in operator coverage by proposing a specialized memory architecture design for also accelerating the not yet supported class of join operators in hardware to a large extent.

## ACKNOWLEDGMENT

We want to thank our industry project partners at IBM Deutschland Research & Development GmbH in Böblingen for supporting our research in this field.

## REFERENCES

[1] J. Gantz and C. Chute, "The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011." IDC, 2008.

[2] "Information Technology Database Languages SQL Part 1: Framework (SQL/Framework)," ANSI/ISO/IEC 9075-1:2008.

[3] W. Inmon, *Building the data warehouse*. J. Wiley, 2002.

[4] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," in *Proceedings of FCCM*, 2012, pp. 45 –52.

[5] P. Francisco, "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics," IBM, Tech. Rep., 2011.

[6] 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, 2010, pp. 1159–1162.

[7] T. Takenaka, M. Takagi, and H. Inoue, "A scalable complex event processing framework for combination of SQL-based continuous queries and C/C++ functions," in *Proceedings of FPL*, 2012, pp. 237–242.

[8] C. Nie, "An FPGA-based smart database storage engine," Master's thesis, ETH Zürich, Department of Computer Science, Systems Group, 2012.

[9] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder: A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS," in *Proceedings of FPL*, 2008, pp. 119 –124.

[10] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," in *Proceedings of FCCM*, 2012, pp. 37 –44.

[11] D. Koch and J. Torresen, "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proceedings of FPGA*, 2011, pp. 45–54.

[12] R. O. Nambiar and M. Poess, "The making of TPC-DS," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 1049–1058.