High-Performance In-Network Data Processing

Jaco Hofmann
Embedded Systems and Applications Group
TU Darmstadt
Hochschulstr. 10
64289 Darmstadt
hofmann@esa.tu-darmstadt.de

Lasse Thostrup
Data Management
TU Darmstadt
Hochschulstr. 10
64289 Darmstadt
lasse.thostrup@cs.tu-darmstadt.de

Tobias Ziegler
Data Management
TU Darmstadt
Hochschulstr. 10
64289 Darmstadt
tobias.ziegler@cs.tu-darmstadt.de

Carsten Binnig
Data Management
TU Darmstadt
Hochschulstr. 10
64289 Darmstadt
carsten.binnig@cs.tu-darmstadt.de

Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
Hochschulstr. 10
64289 Darmstadt
koch@esa.tu-darmstadt.de

ABSTRACT

Recent research has shown the potential for using programmable network components such as switches for distributed data processing. Opportunities include in-network caching and the execution of distributed SQL operations such as joins or aggregations. However, a major weakness of the current generation of programmable switches is that the hardware still has many limitations not only with regard to what type of operations are supported in a switch (e.g., no loops), but also that the switches can often not sustain processing at line-rate.

As a first contribution of this paper, we propose a new switch architecture that can be employed as an in-network co-processor for analytical SQL workloads. Different from existing commercial switches, our switch architecture is based on an FPGA design and supports complex operations at line-rate. As a second contribution, we discuss how a typical distributed database architecture has to be changed to efficiently leverage the new switch architecture. In our evaluation we show that our new switch architecture can significantly speed-up distributed query processing by up to 7× compared to traditional shuffle-based approaches without in-network processing capabilities.

1. INTRODUCTION

Motivation. Scalable database systems for analytical workloads such as Terradata, Microsoft Parallel Data Warehouse, or Amazon's Redshift are being used today for analyzing massive amounts of data in distributed setups. These systems exploit the sheer amount of nodes to leverage data parallelism by shuffling data back and forth. While this paradigm is quite successful, recent papers [7, 1] have shown that data parallelism in a distributed setup does not necessarily lead to improved performance especially in modern main memory databases due to high communication costs or inefficient utilization of the network.

Therefore, many recent papers have suggested to improve the performance of distributed databases by optimizing their network usage with the help of high-speed networks and RDMA [11, 12]. While RDMA allows to leverage high network bandwidth and low latency in database systems, it is not the only option of modern network technologies that can be leveraged by distributed data processing systems. An interesting direction is that network components such as switches are becoming programmable and thus allow to offload processing into the network itself.

This opens up many possibilities for tailoring the network stack to data processing, ranging from opportunities such as in-network caching to the execution of distributed SQL operations inside network components, i.e., in-network processing (INP) [4, 2, 8]. However, especially for in-network processing, a major weakness of the current generation of programmable switches is that the hardware still cannot sustain processing on line-rate and is not capable of many memory intensive operations, such as the computation of SQL joins or aggregations [2, 6] which need to keep an intermediate state.

Contribution. In order to address this challenge, we make the following contributions. First, we propose a new switch architecture that can be used as an in-network co-processor for analytical SQL workloads. The switch architecture relies on a system on a chip (SoC) design that leverages an FPGA and provides larger amount of DDR3 main memory in the switch to execute query pipelines, i.e., sequences of multiple

10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19), August 26, 2019, Los Angeles, California, CA, USA.

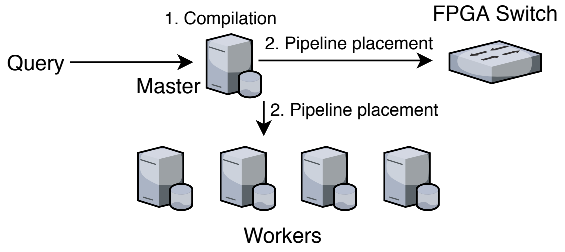


Figure 1: Overview of our Processing Scheme.

SQL operators. By using this architecture we thus cannot only process data at line-rate, but also support more memory intensive operations inside the switch, such as hash-table building and probing.
Second, we show how a typical distributed database architecture can be adopted to efficiently use our new switch architecture. The main idea is that the database comes with a set of pre-compiled query pipelines that are installed in the FPGA. During query compilation the appropriate pipeline is chosen. Furthermore, the query optimizer is extended to determine the best execution strategy of leveraging the in-network co-processor.
Finally, in our experimental evaluation we show that our optimized query processing scheme which utilizes the new switch architecture significantly speeds-up distributed join processing by up to 7×, compared to a traditional shuffle-based approach without INP. In contrast to traditional query processing, our INP-enabled scheme can eliminate the overhead of shuffling intermediate results and thus reduces the communication between nodes in a distributed database.

Outline. The remainder of this paper is structured as follows. In Section 2 we first give an overview of our in-network processing (INP) architecture for analytical SQL workloads and how it can be integrated into a distributed database. Afterwards, we present how query processing and optimization has to change to push SQL operations into the switch and then discuss the design details of our switch architecture in Section 4. Finally, we report initial experimental results in Section 5 and conclude with the limitations of our current prototype and a discussion of future directions in Section 6.

2. SYSTEM OVERVIEW

This section provides an overview of our proposed INP-based query processing scheme and discusses its main differences compared to traditional distributed query processing.

Query Processing. To illustrate the main idea of our distributed query processing that utilizes our proposed switch design, we first review classical distributed query execution and then discuss the changes compared to our scheme. A typical setup of a shared-nothing database consists of one master and several compute nodes, as well as one switch connecting the nodes. As an example query, consider the execution plan in Figure 2 that could result from the SQL statement `SELECT * FROM A JOIN B JOIN C`.
In the classical distributed query processing, `A` and `B` are first shuffled according to the join key. Then, each node builds a hash table over `B` (assuming `B` is the smaller table)

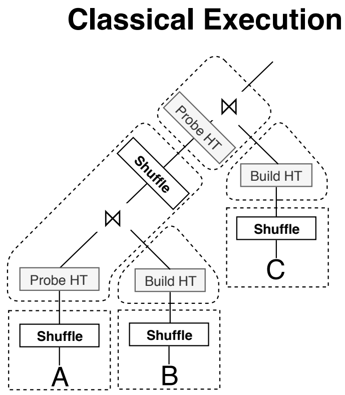


Figure 2: Example of Query Plan for Classical Execution.

and uses tuples from `A` to probe in that hash table. For the subsequent join, the intermediate result of $A \bowtie B$ as well as relation `C` need to be shuffled again, such that the joins can be executed by building and probing into the hash table of `C`. Thus, each join (if data is not co-partitioned) typically requires one expensive shuffle operation.
In order to avoid the repeated shuffling of data, we propose a new execution scheme that utilizes our proposed switch for in-network processing. Figure 1 shows the two main steps of our scheme. As a first step, the master node determines and compiles an optimal execution plan when a new SQL query arrives (Figure 1 ①). Next, the master node places the different steps of the plan on the worker nodes as well as the switch (Figure 1 ②). How an optimal execution plan for our switch architecture is created and how the pipeline placement is performed will be described in more detail in the next section.
The equivalent query plan for INP execution is shown on Figure 3. For now, assume that the plan is the optimal one for our example query. As can be seen in the two figures, the classical - Figure 2 and the INP-based plan Figure 3 consist of two types of pipelines (probe-pipelines and build-pipelines), the INP-based plan splits the plan into multiple pipelines that can be placed on worker nodes or the switch respectively. As a consequence, different from the traditional plan many of the `Shuffle` operators can be completely avoided since the probe steps are executed all in the switch. In the following we will discuss the implications of the differences in more detail.

Discussion. As mentioned before, the main conceptual difference of our scheme is the elimination of shuffling, and in particular the re-shuffling of intermediate join results. This is beneficial, since shuffling comes with several challenges.
First, shuffling operations are so called pipeline breakers, since the streaming of tuples through an operator pipeline is stopped (i.e., the shuffle operation only starts once the previous intermediate result has been materialized completely). This however, limits the degree of parallelism of the execution since following phases of a query need to wait for the completion of previous ones. For instance, the second join of our example query can not be computed until the result of the first join has been materialized.
Second, shuffling usually means that significant amounts of data need to be transferred via the network, since also

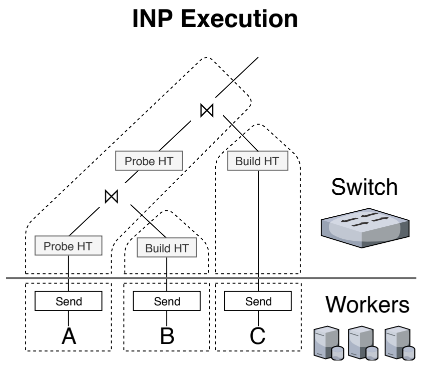


Figure 3: Example of Query Plan for INP Execution with Operator Placement.

the intermediate results need to be partitioned and sent to all workers. The cost of shuffling intermediate results is even higher in a data warehouse setup. This is because, in a star schema with one very big fact table and multiple smaller dimension tables that need to be joined, the cost of shuffling the fact table and the resulting intermediate results is dominating the overall query execution cost. Considering the example query plan shown in Figure 2, the fact table could be represented by relation `A` and the dimension tables by `B` and `C`.

Finally, we are less sensitive to skew, since typically one node receives more data than the others. When multiple nodes send to a single node, the network link of the node gets congested and slows down overall execution (also known as incast problem).

3. QUERY PROCESSING

In this section we describe how the database architecture can be adapted to leverage the FPGA-based switch.

3.1 Query Compilation

The query compilation resembles a physical execution plan for a given query. One important decision in distributed systems during optimization is *where* to execute pipelines optimally. Consequently, our adapted query compilation takes the FPGA switch as a processing unit into account. When employing an FPGA for query processing, it is not feasible to synthesize a complete configuration (a so called bitstream, i.e., the executable logic on the FPGA) on a query-to-query basis, as bitstream generation can take multiple hours. However, once the bitstream is generated and installed on the switch FPGA, re-configuring the switch to use a different pre-installed bitstream only takes a few milliseconds. Hence, our system allows to install a set of bitstreams for pre-generated pipelines to execute multiple different queries efficiently.

Figure 3 shows a physical operator plan for our example query. As shown, each worker is only responsible to send its part of the relation to the switch, which executes the main query pipelines, i.e., building and probing pipelines for executing joins. To support generic queries inside the switch, the pre-generated pipelines provide different signatures. For instance, the intermediate hash table for table `B` needs to store keys and values of 8 Bytes, whereas table `C` needs 4 Byte keys and 10 Byte values.

The master node thus tries to choose the best fitting signature, if there is no exact match it takes the next larger one. This clearly induces memory overhead, e.g., if the relation has a 64 Byte value, then the master chooses the 128 Byte pipelines. However, this should not be a common case, since optimal signatures can be generated as soon as the workload is known.

3.2 Query Optimization

In a traditional database system, the optimizer is responsible for finding the best plan. INP has a slightly different execution model and mandates an extension to the existing cost-based optimization. We therefore propose the following optimization objective: The optimizer should reduce the number of re-shuffle operations by offloading computation to the switch. These pipeline-breaking operations are especially expensive, since they require synchronization until the query execution proceeds. Moreover, the limited memory of the switch has to be taken into account. Therefore INP should be applied for the most beneficial joins and the optimization problem is in fact a constrained optimization problem.

In our prototypical implementation we only consider left-deep join trees with primary-foreign relations as in Figure 2 for INP, since this is a common join structure in analytical workloads.

In the following, we first explain our notation and then derive our cost model. We are considering a left-deep plan which consists of a left-deepest relation L (L might be an input relation or an intermediate result) and a set of tables T_i which are joined with L. The number of workers is defined by N. $|R|$ denotes the cardinality of any relation R and $ts(R)$ the size of a tuple in R. The subset of tables indexed by I which are qualified for INP is defined as $L \bigcup_{i=1}^{I} T_i$. For instance the query plan shown in Figure 2 consists of the tables A, B, C. However, due to high memory requirements, or lower costs of a shuffle-based join, the optimizer could define I such that only A and C are joined with INP. Consequently, the intermediate result of $A \bowtie C$ would be joined with B with the shuffle-based approach.

Cost Model for Classical Model. Based on the network cost of one relation, we describe the network cost for the classical approach used in distributed databases which is based on data shuffling. Later, we derive a new cost model for our INP-based query processing scheme. The following equation describes the cost for sending the qualifying tuples of one relation over the network:

$$c_{rel}(R) = |R| * ts(R) \tag{1}$$

In the classical cost model, the cost to create an intermediate join of the tables indexed by I is thus given by the following equation:

$$\begin{aligned} c_{\mathit{shuffle}}(I) = \frac{N-1}{N}\Big(& c_{rel}(A) + c_{rel}(A \bowtie T_{i1}) + \dots \\ & + c_{rel}(A \bowtie T_{i1} \bowtie \dots \bowtie T_{im-1}) + \sum_{i \in I} c_{rel}(T_i)\Big) \end{aligned} \tag{2}$$

To explain the above equation we calculate the cost for the plan shown in Figure 2, with four workers, and the following parameters.

Relation	Size ($\|R\|$)	Tuple Size (ts(R))	$c_{rel}(R)$
A	100.000	32	3.200.000
B	10.000	10	100.000
C	10.000	10	100.000
AB	100.000	32	3.200.000

We consider the complete plan in our example - the intermediate result consists of A, B, C, therefore the costs of A, B, C can be included in the Equation (3). These describe the cost for shuffling A, B, C. Besides these input relations, we also shuffle the intermediate result of $A \bowtie B$. Hence, we include those costs as well, resulting in:

$$c_{shuffle}(I) = \frac{N-1}{N} \left(3.2 \times 10^6 + 3.2 \times 10^6 + 100 \times 10^3 + 100 \times 10^3\right) \tag{3}$$

Finally, we assume that only $\frac{3}{4}$ of the data is shuffled in the uniform case w/o skew and $\frac{1}{4}$ is kept locally. Consequently, the network cost for the given plan is $\frac{3}{4} * 6.6 \times 10^6$.

Cost Model for INP. Now, the INP cost model can be derived from the classical model. The major change is to avoid re-shuffling of intermediates and thus they are removed from the equation. Hence, the following equation holds:

$$c_{INP}(I) = \left(c_{rel}(L) + \sum_{i \in I} c_{rel}(T_i))\right) \tag{4}$$

As shown in the equation, the network cost of the INP approach is determined only by the tables and tuples sizes and not by the intermediate results. We again apply this cost function to the example above.

$$c_{INP}(I) = \left(3.2 \times 10^6 + 100 \times 10^3 + 100 \times 10^3\right) \tag{5}$$

This gives the following network costs for the INP approach 3×10^6.
Hence, the cost improvement of the classical approach vs. the INP based approach is $c_{INP}(I) - c_{shuffle}(I)$ depending on the set of tables used to apply INP. The optimization problem is now to choose $I \subset \{1, 2, \dots, n\}$ such that this improvement is maximized while the memory constraint in the switch $\sum_{i \in I} |T_i| \leq C_{RAM}$ is satisfied. Note that also $I = \emptyset$ is considered in this optimization problem, i.e. INP is not applied at all.

Theoretical Analysis:. This paragraph discusses the previously introduced cost-models and elaborates when INP is beneficial. Note, however INP is not a complete replacement of the traditional approach, but rather an optimization which can be used for some cases. We first show the effect of the network cost $c_{rel}(A)$ from A in relation to the other tables T_i. Based on Equation (1) the cardinality and the size of the tuples have an impact on the network cost of A. To analyze the effect of increasing costs of the left deepest relation we again use our query plan from Figure 2 with four workers. Figure 4 shows on the x-axis the network cost of $c_{rel}(A)$ in relation to B, C, i.e., 0.1 means that the cost of the relation A is only 10 percent of B, C. To show the effect in isolation we

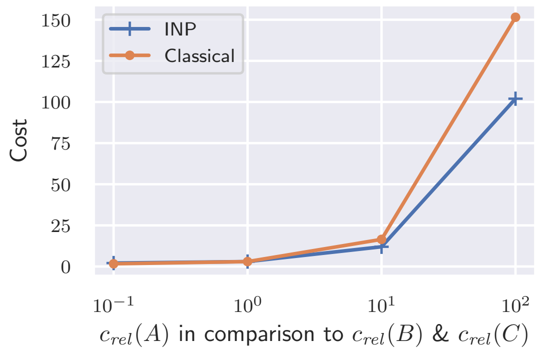


Figure 4: Cost analysis for different table ratios. With bigger relation `A` in comparison to `B` & `C`, the INP approach greatly reduces cost.

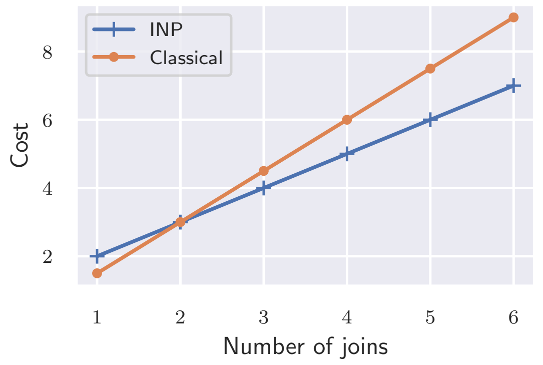


Figure 5: Cost Analysis for varying number of joins. Relation costs (c_{rel}) are kept the same for all joined relations.

thus fix B, C to the same size. The y-axis shows the outcome of the cost functions for $c_{shuffle}(A, B, C)$ and $c_{INP}(A, B, C)$. The plot shows that if the costs for A is smaller than the cost for B, C, the classical approach is more suitable. Since the intermediate results will be cheaper, shuffling only a fraction of our relation $(\frac{N-1}{N})$ is cheaper than sending everything to the switch. However, INP is more efficient if the cost of A exceeds B, C, thus we avoid expensive reshuffling on intermediate results.
The best performing strategy is not only determined by the costs of A, B, C. The number of joins also influences the decision. Therefore, the next plot analyzes the effect of the number of joins when A and T_i have equal costs. Figure 5 shows on the x-axis the number of joins and the y-axis shows again the costs of the two strategies.
By avoiding the intermediate shuffle, the INP approach is clearly beneficial if the number of joins increases. In conclusion we have shown that the INP approach is beneficial if the cost of A is high compared to the other relations and further if the number of joins is high. Both of these circumstances are often met in data warehouse scenarios.

Cost-Model Extensions. This paragraph extends the proposed cost-model to support selections, co-partitioning, and skew. *Selections* can be modeled by multiplying $c_{rel}(R)$ with

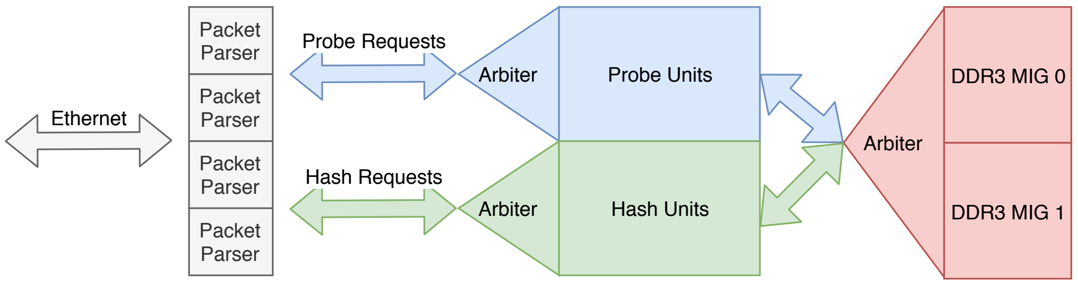


Figure 6: Overview of the proposed architecture on the NetFPGA SUME board. Data is processed as a stream of 64 bit words provided by the Xilinx 10G Ethernet Subsystem. The Ethernet packets are parsed using a Bluespec-generated packet parser. The extracted hashing and probing requests are forwarded to the hashing and probing infrastructure.

the selectivity. This not only allows to support selection predicates but other types of joins as well. *Co-partitioning* is only relevant to the classical approach and is modeled trivially by removing the relations from the equation. Assuming that A, B are co-located in our example query, they can be joined locally and there is no need to shuffle A, B. However, it is still necessary to shuffle $A \bowtie B$ and C. In the INP approach, all tables need to be sent to the switch, and consequently co-partitioning has no effect on the costs. *Skew* is modeled by applying a write amplification factor to the costs of $c_{shuffle}(I)$. Furthermore, skew often leads to the incast problem; i.e., the ingoing link on the worker receiving the large amount of skewed data becomes congested and acts as a bottleneck. Hence, the write-amplification factor models that skew since it increases the shuffle cost. In contrast, the INP approach is not affected by skew since in the best case data does not need to be shuffled at all.

4. SWITCH DESIGN

In the following, we describe the design of our switch architecture that can be optimally used as an in-network co-processor for typical analytical SQL workloads. We first explain the hardware platform our switch is based on before we explain how different query pipelines for hash table building and probing are supported inside our switch architecture.

4.1 Hardware Platform

The platform chosen for the demonstrator is the NetFPGA SUME [13], based around a Xilinx Virtex 7 FPGA, 8 GB of DDR3-SDRAM memory and four SFP+ connectors. Those connectors can be used to interface the FPGA with off-the-shelf SFP+ solutions common in data centers, either via fiber optics or direct-attached cables. The bandwidth of all ports is 10 Gbit/s.

The switching hardware itself is described in Bluespec SystemVerilog, a Hardware Description Language (HDL) that combines high-level features of functional programming languages with the performance of hand-crafted low-level HDLs such as SystemVerilog or VHDL.

This section describes the different stages of packet processing in the proposed architecture: (1) Packet Parsing, (2) Hash Table Generation, and (3) Hash Table Probing.

For designing our switch architecture we leverage the TaPaSCo [5] tool chain. The tool chain provides all necessary steps to bring hardware acceleration to a variety of platforms, in many cases avoiding the need for explicit hardware development knowledge. The tools assist with all necessary steps such as bitstream generation, bitstream loading and interfacing to a host computer for control and monitoring tasks. TaPaSCo assists the designer finding the optimal working conditions for a given architecture. Furthermore, TaPaSCo has already been employed succesfully in other in-network-processing applications [3].

4.2 Ethernet Packet Parsing On FPGA

The interface to the SFP+ connectors is provided by Xilinx through their 10 Gigabit Ethernet Subsystem IP core. The packets received over SFP+ are provided by the core as a stream of 64 bit words at 156 MHz.

The streams of all four interfaces are collected in their corresponding packet parser infrastructure in the proposed architecture, as shown in Figure 6. These packet parser units can operate at line rate and are completely independent from each other. The packet stream is parsed using a custom parsing state machine generator. Common functionality such as dropping packets for the wrong destination MAC address or with the wrong protocol is done on the fly as soon as the relevant data is available.

Requests, in our case the probe or hashing modes, are immediately forwarded out of the parsing module for further processing. These taps into the parsing pipeline can occur at any processing step and provide very flexible protocol handling. Previous parts of the packet can also be examined in later steps by explicitly marking certain parts of the packet as relevant for later processing.

The extracted requests are then stored in FIFO buffers to be collected by the hashing and probing infrastructure for actual processing.

4.3 Hash Table Generation

The example application demands very high hashing performance from the system. Each of the 10 Gbit ports results in over 38×10^6 hash table inserts per second. The architecture should support this throughput for all pipelines in parallel. Latency, on the other hand, is not important as an insert does not result in any feedback to the sender. Accordingly, the sender will simply send out the pipelines without waiting for ACK signals or similar. The hash table generation architecture supports only insert operations, as deletes are not required, and probes are handled by a different part of the design.

Key Value Stores on FPGA are a well-researched field [9, 10]. These approaches usually differ from CPU based hash table implementations as FPGAs have different strength and weaknesses. For example, the FPGA can process wider words,

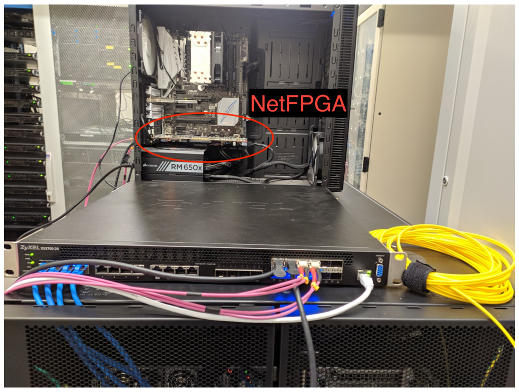


Figure 7: Setup used to evaluate the proposed architecture. Four nodes with Xeon 5120 CPUs are connected via 10 Gbit/s links to a central Zyxel XS3700 switch. The NetFPGA SUME-based switch is connected with two SFP+ Fiber and two Direct Attached cables. The FPGA is installed in a host PC for simplified monitoring via PCIe. The host only performs management tasks, and thus does not process any packets on its own.

such as 512 bit whereas a CPU based implementation has to consider the caching infrastructure of the given processor.
The design proposed here works in the following way:

1. Hash the key provided in the request by the parsing stage to calculate a bucket.
2. Retrieve the corresponding bucket from the main memory. All buckets are 512 bit in size, which corresponds to the data path width of the DDR3 controllers.
3. Place the key and value tuple in the first free position in the bucket and write the bucket back to main memory.

All of these stages are pipelined and multiple requests are processed at any time. The pipelining might result in Read-after-write hazards which are dealt with by the look-ahead buffers. These buffers inject the answer read from the memory, including the newly added tuple, whenever a succeeding insert request hashes to the same bucket.
To spread the hashing load over memory as much as possible an interleaved approach is used. Each of the hashing units uses all of the available memory but only every third entry belongs to a certain unit. For example address 0 stores bucket zero of hash table zero, address 64 stores bucket zero of hash table one and address 128 stores bucket zero of hash table two. This scheme allows for much better utilization of all the available memory resources compared to a simple block-wise arrangement.
The performance of this architecture is completely determined by the random access Read/Write performance of the DDR3 controllers and is typically around 56×10^6 inserts per second per memory controller. Higher performance can be reached by utilizing newer devices with memories such as HBM having higher random access speed.

4.4 Hash Table Probing

Compared to the table generation, probing has to meet even higher performance requirements. For every probing request, all stored hash tables have to be probed in parallel. For one 10 Gbit/s link and three hash tables this combines to about 120×10^6 requests per second in total.
The architecture itself closely resembles the design of the insert units but without the write-back part. Every lookup requires three steps:

1. Hash the key provided in the request by the parsing stage to calculate a bucket.
2. Retrieve the corresponding bucket from the main memory.
3. Return the key contained in the bucket, or an invalid flag if the key is not found.

Again the performance is completely determined by the performance of the DDR3 controller. Considering that only reads, and no writes, are necessary the performance is about 75 % better at around 98×10^6 probes per second per memory controller.
The results of the probes are forwarded to a combination unit which is responsible for answering the probe requests. The retrieved tuples are combined with the original request and forwarded to the SFP+ parsing units which in turn send out the completed requests to their destination.

4.5 Performance

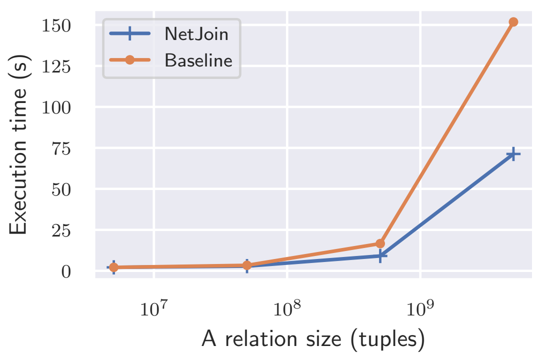
The architecture is shown to be able to handle packet processing at line rate and even multiple channels in parallel. The hashing mechanism is able to handle around 30 Gbit/s of traffic and is only limited by the available random access performance of the memory controllers. The probe units are able to process up to 197×10^6 tuples per second using two DDR3 controllers.

5. INITIAL RESULTS

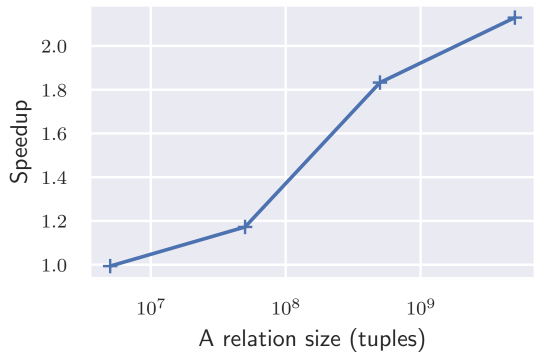
In the following, we show the initial results of our new switch design in a distributed database.

5.1 Setup and Workload

Our experiments were executed on a five node cluster – one master node and four compute nodes. Each server has an Intel(R) Xeon(R) Gold 5120 CPU @ 2,20GHz processor and 384GB RAM, running Ubuntu 18.04. All machines were equipped with a 10 Gbit/s NIC. The four compute nodes are connected via CAT 6 RJ45 Ethernet Cables to a Zyxel XS3700 switch (without INP) and our FPGA-based switch (with INP capabilities). The FPGA switch is attached to our compute nodes using two SFP+ DAC cables from Digitus and two SFP+ fiber transceivers by *FLEXOPTIC*. A picture of our experimental setup is shown in Figure 7.
Based on this setup, multiple experiments were conducted to demonstrate the performance of our proposed architecture (referred to as *NetJoin*) over a baseline without INP. The experiment represents a shuffle-heavy scenario like described in Section 2. A table `A` is joined together with three other tables `B`, `C` & `D`. The join shows similarity to a data warehouse setup with `A` being the fact table with foreign keys to the dimension tables `B`, `C` & `D`.
In our setup, all tables are pre-partitioned such that no join partners can be found locally without the need of sending one of the tuples over the network. In data warehousing, typically one of the dimensional tables in a star schema can be co-partitioned with the fact table and thus shuffling can be avoided for the first join in a plan where the co-partitioned dimension table is joined. However, this optimization can be applied for both processing schemes; the traditional shuffle-based and our INP-based scheme and both would benefit equally. In our experiments, we thus show the performance of both schemes without this optimization.

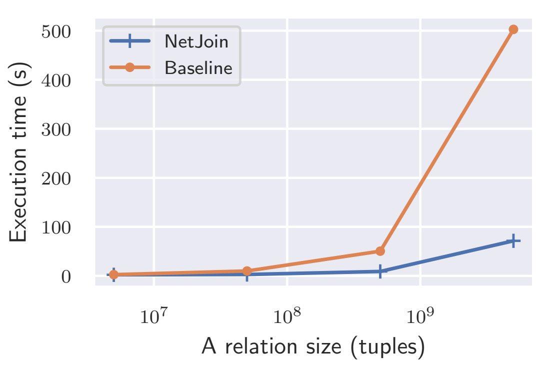


(a) Join runtime.

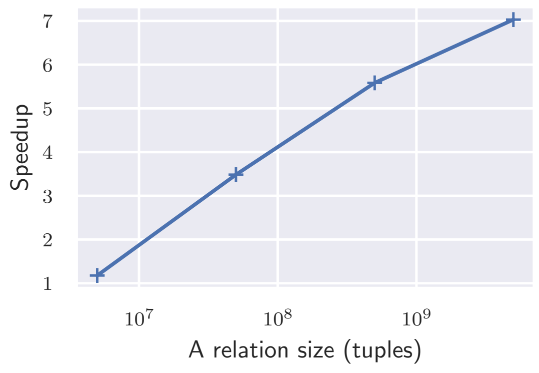


(b) Speedup of *NetJoin* over baseline.

Figure 8: Experiment 1. Four nodes joining ranging A relation sizes (5×10^6 to 5×10^9 tuples) with fixed B, C & D relations (50×10^6 tuples). Link speed on each node at 5 Gbit/s. For small sizes of A, the shuffling overhead is too small to make a significant impact on runtime. Larger sizes of A, compared to relations B, C & D, result in increased overhead due to the required shuffling. Accordingly, *NetJoin* is increasingly faster compared to the baseline as it does not need to shuffle the tables.



(a) Join runtime.



(b) Speedup of *NetJoin* over baseline.

Figure 9: Experiment 2. Shuffle skew on four nodes joining ranging A relation sizes (5×10^6 to 5×10^9 tuples) with fixed B, C & D relations size (50×10^6 tuples). Link speed on each node at 5 Gbit/s. *NetJoin* executes with same runtimes as compared to the unskewed scenario shown in Figure 8(a). The baseline however is impacted by the bottleneck resulting from the non-uniformly distributed join keys.

Finally, a last important fact is that the *NetJoin* makes use of raw Ethernet frames, such that higher level protocols do not split up packets and no performance overhead is introduced. In our paper (as well as other INP papers such as [6]), we do not yet handle dropped frames. Instead, throughout the experiments we made sure that the amount of dropped tuples are monitored and limited to at most 2%. For providing reliability, a light protocol could be implemented on top of raw Ethernet which does not introduce a significant performance penalty.

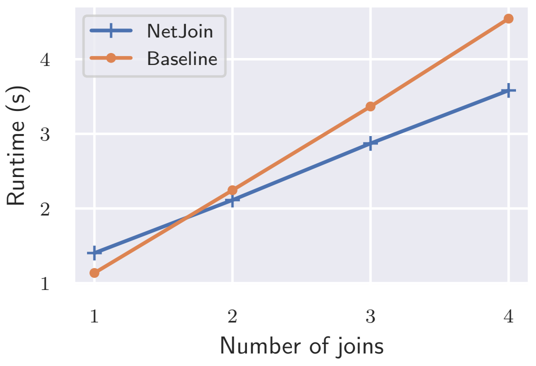
5.2 Experiment 1: Uniform Join Keys

The first experiment shown in Figure 8 scales the size of the A relation in comparison to relations B, C & D. The left graph (a) shows the runtime of the distributed hash join over the varying sizes of the A relation. The B, C & D relations sizes are 50×10^6 tuples, and with the A relation ranging from 5×10^6 to 5×10^9 tuples. Since the join keys are uniform, each of the four nodes receive the same amount of tuples when shuffling the relations.

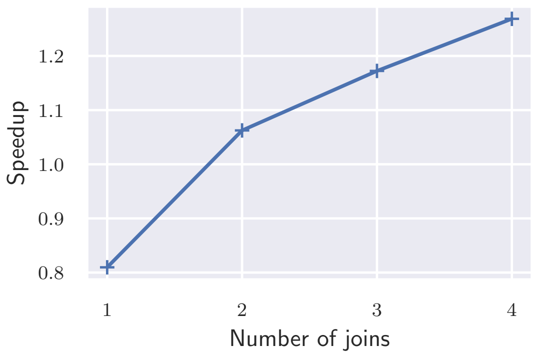
The results show that as the A relation size is small, the *NetJoin* does not perform better than the baseline since reshuffling the intermediate results is inexpensive due to A's small size. As the A relation size grows, the *NetJoin* outperforms the baseline. Even though the nodes in the *NetJoin* have to completely send their local partitions of all relations to the switch, the reduced cost of reshuffling compensates for this. The speedup as shown in Figure 8(b) shows an over 2× performance gain against the baseline with the A relation 100× bigger than B, C & D.

5.3 Experiment 2: Skewed Join Keys

To show that our in-network execution scheme is more resilient against skew compared to a traditional query processing, we generated skewed join keys. The skew was such that when shuffling the relations on four nodes, 80% of all tuples go to Node 1, 13% to Node 2, 5% to Node 3 and the remaining 2% to Node 4.



(a) Join runtime.



(b) Speedup of *NetJoin* over baseline.

Figure 10: Experiment 3. Scaling number of executed joins in query from 1 to 4. All relation sizes are fixed to 50×10^6 tuples. *NetJoin* is slower for 1 join since the complete relations are sent to switch. With more joins *NetJoin* outperform the baseline through not having to shuffle intermediate joined relations. With relation `A` bigger than joined relations, the speedup increases further as demonstrated in Experiment 1.

Since the initial prototype only uses raw Ethernet frames, conducting this experiment for the baseline meant that Node 2, 3 & 4 needed to throttle the speed of outgoing packets to Node 1. This ensured packets were not being dropped due to congested in-going link to Node 1.
As shown in Figure 9(a) such skewed shuffling scenario heavily affects the performance of a distributed join, not only because the compute intensity and memory consumption are not equally distributed, but also because of incast congestion in the network switch. Since Node 2, 3 & 4 all need to send 80% of their local relation to Node 1, the in-going link is acting as a bottleneck and other nodes throttle down their sending rate.
However, with our *NetJoin*, skew on the join key does not play a role since no network shuffling is taking place. Figure 9(a) shows an identical runtime of the *NetJoin*, but with the baseline performance severely suffering in comparison to Figure 8(a). The speedup shown on Figure 9(b) reports a speedup of 7× for the largest `A` relation size.

5.4 Experiment 3: Scaling Number of Joins

As discussed in Section 3, not only the relation size of the left deepest relation, but also the number of joined relations has an impact on the query runtime. In this experiment we show that our proposed approach is superior to the classical query processing when the number of joins increases. This is also true for the sub-optimal case when the size of the left deepest relation is not larger than the other relations (i.e., the fact table and dimension tables have the same sizes).
The experiment is executed by fixing all relation sizes to 50×10^6 tuples. Figure 10(a) shows the runtime of queries with 1 to 4 joins. As already shown in the cost analysis in Section 3 (Figure 5), the baseline cost increase with a higher gradient than the cost of *NetJoin* (INP). Moreover, we can see that only after two joins the INP-based approach outperforms the classical approach.
Additionally, we also conducted an experiment where the fact table is larger than the dimension tables. In this case, the INP-based approach again outperforms the classical shuffle-based approach by a higher factor.

6. CONCLUSION & FUTURE WORK

This work is motivated by the observation that existing programmable switches cannot process memory intensive operations and thus are not suited for distributed query processing.
To overcome this issue, we proposed a new FPGA-based switch architecture. By using an FPGA-based design we are flexible to process different incoming queries at line-rate. The high-level FPGA programming paradigm used here provides considerably more flexibility than existing solutions, such as P4's match+action stages.
Furthermore, we also discuss initial directions where to adapt distributed query processing to leverage the capabilities of INP. The main idea is to avoid expensive shuffling operations by offloading more complex query pipelines to the switch. We show that our proposed execution scheme can thus speed up query processing for left-deep join plans by up to 7×.
However, our prototype has also shown some limitations that we could not yet address. We leave these limitations for future work. One of the limitation of the current design is that the output of a probing pipeline cannot be larger than its input due to congestion. By using a more recent FPGA board to realize the switch, up to 256 GB of DDR4-SDRAM plus 8 GB of very fast HBM will be available, thus allowing the implementation of a better caching scheme. This cache could allow us to better control the congestion of the outgoing link by buffering/stalling some tuples before sending out.
Finally, while the initial results of our prototype are promising, there are many other open routes for future work that we have not been able to address yet. For example, in a real data-center setup multiple switches are involved. To that end, we could use parallelism in the network by using multiple of these switches. Furthermore, another direction would be fault-tolerance or isolation of multiple queries which are all not yet handled in our current design.

Acknowledgements. This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 (DFG priority program 2037) as well as the DFG Collaborative Research Center 1053 (MAKI).

References

[1] *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, 2016. IEEE Computer Society. ISBN: 978-1-5090-2020-1. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7491900`.

[2] M. Blöcher, T. Ziegler, C. Binnig, and P. Eugster. Boosting scalable data analytics with modern programmable networks. In *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, 1:1–1:3, 2018.

[3] T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, and F. Pedone. Consensus for non-volatile main memory. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[4] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure accelerated networking: smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66, 2018.

[5] J. Korinth, J. Hofmann, C. Heinz, and A. Koch. The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems. In *International Symposium on Applied Reconfigurable Computing (ARC)*, 2019.

[6] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The case for network accelerated query processing. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[7] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[8] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 150–156, 2017.

[9] D. Tong, S. Zhou, and V. K. Prasanna. High-throughput online hash table on fpga. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 105–112, May 2015. DOI: `10.1109/IPDPSW.2015.149`.

[10] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang. Memory efficient and high performance key-value store on fpga using cuckoo hashing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug. 2016. DOI: `10.1109/FPL.2016.7577355`.

[11] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: distributed transaction can scale. *PVLDB*, 10(6):685–696, 2017. DOI: `10.14778/3055330.3055335`. URL: `http://www.vldb.org/pvldb/vol10/p685-zamanian.pdf`.

[12] T. Ziegler, C. Binnig, and U. Röhm. Skew-resilient query processing for fast networks. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, pages 81–85, 2019.

[13] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept. 2014. ISSN: 0272-1732. DOI: `10.1109/MM.2014.61`.