The Case for Network-Accelerated Query Processing

Alberto Lerner Rana Hussein Philippe Cudre-Mauroux
eXascale Infolab, U. of Fribourg—Switzerland

ABSTRACT

The fastest plans in MPP databases are usually those with the least amount of data movement across nodes, as data is not processed while in transit. The network switches that connect MPP nodes are hard-wired to perform packet-forwarding logic only. However, in a recent paradigm shift, network devices are becoming "programmable." The quotes here are cautionary. Switches are not becoming general purpose computers (just yet). But now the set of tasks they can perform can be encoded in software.

In this paper we explore this programmability to accelerate OLAP queries. We determined that we can offload onto the switch some very common and expensive query patterns. Thus, for the first time, moving data through networking equipment can contribute to query execution. Our preliminary results show that we can improve response times on even the best agreed upon plans by more than 2x using 25 Gbps networks. We also see the promise of linear performance improvement with faster speeds. The use of programmable switches can open new possibilities of architecting rack- and datacenter-sized database systems, with implications across the stack.

1. INTRODUCTION

Networking is an area in constant evolution. New protocols keep arising from emerging fields such as virtualization [20], cloud computing [6], or the Internet-of-Things [27]. Many such protocols are implemented in hardware-based switches. In the past, support for a new protocol would require a new version of a switching silicon – something very costly and time consuming. Recently, however, chips such as Barefoot Tofino [1], Cavium Xpliant [2], and Cisco Quantum Flow [3] started to support programming protocols via software.

At the core of this innovation is a packet-processing hardware called *Match-Action Unit* (MAU). A MAU combines a *match* engine with an *action* engine. The *match* engine holds data in a table format and can match a packet's fields with a row in this table using, for instance, exact matching. Other types of matches are also possible. The *action* engine executes simple instructions over a packet or table data. Examples of such instructions are simple arithmetic or moving data within a packet. The MAU is programmable in the sense that one can specify its table layout, the type of lookup to perform, and the processing done at a match event, as we illustrate in Figure 1(a). We say that a MAU implements a *match-action table* (or, simply, a *table*) abstraction.

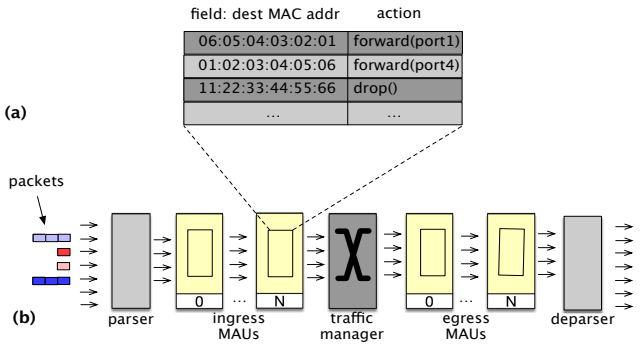

Figure 1: (a) A match-action table programmed to forward or to drop a packet according to its destination MAC address. (b) Architecture of a programmable switch dataplane holding that table.

Such a table abstraction is powerful enough to express very common computations in networking protocols. For example, it can encode many variations of IP lookup [26] or packet classification [15] – two of the most recurring problems in packet forwarding. To support full protocols, several MAUs can be combined in a pipelined fashion to form a *programmable dataplane*. The dataplane is complete with a programmable packet parser/deparser [13] and a traffic manager (e.g. a buffered, routing element that moves packets across switch lanes). We depict such a dataplane in Figure 1(b). Some tables may use fields from the very packet routing decision made by the traffic manager. Therefore, there are MAUs in both the *ingress* and the *egress* sides of the routing element. Incidentally, the traffic manager itself can also be a programmable. [25].

From a database perspective, the switch was historically a passive element, routing packets for networking purposes. From a networking perspective, tuples generated by query execution were opaque payload. We argue here that a pro-

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

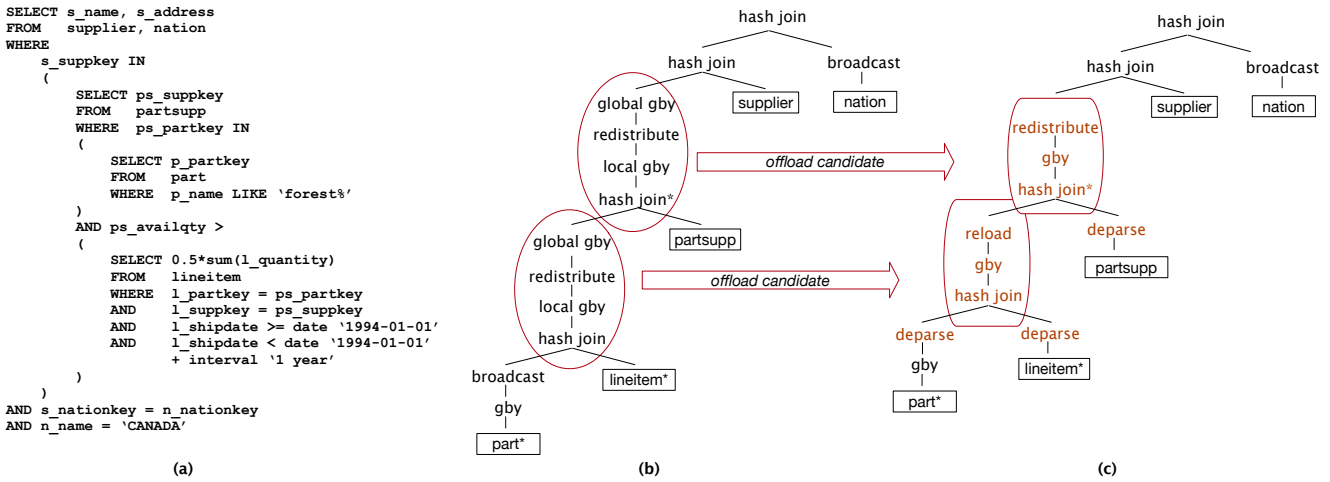

Figure 2: TPC-H Query 20 is shown in (a). Its best plan, one that minimizes data motion [24], is depicted in (b) in a simplified way. (Operators with asterisks perform scan and filtering.) We note that there is a distinct pattern that appears twice in the plan. We show in (c) how to offload that pattern onto the switch by altering the query plan and using special operators.

grammable dataplane can be a bridge to better integrate these two views.

Motivating Example: Consider TPC-H Query 20, listed in Figure 2(a), stripped of its order-by clause. The query is essentially a join of five relations, usually partitioned across MPP nodes. The best agreed upon plan, shown in Figure 2(b), chooses a join order to minimize data motion generated by the `broadcast` or `redistribute` operators [24].

A particular operator pattern is used twice in that plan – a join, followed by a *local* group-by, a tuple redistribution, and a *global* group-by. We show in Figure 2(c) that some segments of the query could be *offloaded onto the switch* by introducing an alternative query plan.

The new plan uses a new operator called `deparse` instead of the traditional data motion operators. This operator is responsible for enqueuing tuple data for transmission, which will then be manipulated by the switch. Furthermore, the new plan deploys an arrangement of MAUs on the switch to store and update the intermediate state of the join and group-by operators (*i.e.*, their hash tables), effectively implementing specialized versions of these operators on the dataplane.

To illustrate one such operator, consider a hash-based equi-join. It builds a hash table over the *outer* relation of a join using a (MAU) table. The MAU is programmed to match the relevant *inner* relation tuples against its table. The action upon finding a match would be to annotate the inner tuple as having passed the join condition. A default action would be to mark that tuple to be dropped. In practice, there are a number of aspects to consider. How are the specific tuples that participate in this join to be recognized inside a packet? How is the state of the operator, its hash table, built up? What if it does not fit in the available space? To address these aspects respecting the peculiar programming model imposed by MAUs, we redesigned basic query operators for this environment from scratch.

For the first time, entire segments of a query plan can be performed on the switch, with potentially strong consequences to data placement, query execution and optimization. The tuples are processed on the switch at "line-speed" – up to 100 Gbps in our case – yielding performance improvements that increase with network speed.

Contributions: The main contributions of this paper are as follows:

- We assess the viability of using programmable switches as MPP nodes. This effort is orthogonal – and can coexist – with other attempts to rethink distributed query execution on high speed networks [8, 9, 28]. To the best of our knowledge, this is the first work incorporating programmable network devices in a SQL query execution setting.
- Even if we have an optimistic lookout, several areas of further research are necessary. The second contribution of this paper is to broadly map such areas.

In the remainder of the paper, Section 2 gives a brief overview of programmable switches. We propose a network-accelerated MPP system in Section 3, and describe a prototype database implementing the main aspects of this system in Section 4. Next, we discuss issues raised by the use of this technology in Section 5, and explore a number of potential research avenues in Section 6. Finally, we discuss related work in Section 7 before concluding in Section 8.

2. BACKGROUND

A MAU is carefully architected to spend a fixed amount of time per packet only, as it must sustain the determined bit rate of the protocol being run – e.g., 100 Gbps Ethernet. To do so, a MAU must be able to realize table matches in constant time. If a table requires only exact matches, a MAU usually resorts to SRAM storage and matching via hashing. More advanced matching techniques such as longer-prefix match rely on *Ternary Content Addressable Memories* [29], or TCAMs, to deliver similar constant-time lookups. Different MAU implementations exist. Some examples include the *Reconfigurable Match Tables* [11], with its variants *Disaggregated RMT* [12], and *FlexPipes* [22]. Each uses a different combination of memory types and arrangements to support matches.

Action engines are also designed with deterministic speed

in mind. They achieve so by limiting the number of instructions that can be issued per match (though they allow instructions to run in parallel). It is common to structure the action engine as a Very Long Instruction Word (VLIW) machine.

A programmer is shielded from these implementation details by coding at the match-action table abstraction level. One language that allows such coding is `P4` [10]. A `P4` compiler can analyze dependencies among tables and decide how to best allocate tables to MAUs [17]. `P4` resembles `C` in its structure, but had a few new constructs added and quite some features suppressed. Although the language itself is quite simple, a variety of challenges arise from its programming model. Almost all of these challenges can be traced back to two aspects.

First, compared to a regular CPU, a programmable dataplane requires specific code placement. A program needs to separate clearly the code for parsing from code for a table. Moreover, a table is specifically designed to operate on either the ingress or egress side of the pipeline. Certain hardware units are only available to certain sections of the code. For instance, there are typically no Arithmetic Logic Unit inside a packet parser. Therefore, arithmetic operations in that section of the code are not supported. `P4`'s syntax embodies those restrictions.

The second interesting aspect is that there is no fast or slow program on a switch. In a sense, a `P4` program is a description of an assembly-line protocol through which each packet goes. That assembly line may use more or less steps – meaning the latency varies among programs – but the conveyor belt runs at a predetermined, constant speed built into the hardware. We say that the code must work at *line speed*. For this reason, loops and other constructs such as dynamic resource allocation that make a program's runtime non-deterministic are not present in the language.

`P4`'s limitations allow a compiler to verify, for a given target, that a program can run there properly. A program may be capable of running at line speed but the hardware may not have enough MAUs to run it. (Currently, we know of dataplanes with 12 to 20 MAUs.) The compiler would flag such a situation. Another reason for a program to fail is allocating tables that are larger than even what a set of MAUs could hold. (MAUs that offer 0.5 MB of SRAM, for example, can be found in some current models.)

As we discuss next, in order to take advantage of a programmable switch, we need to revisit a number of aspects of classical database architecture.

3. NETACCEL: A NETWORK-ACCELERATED MPP DATABASE SYSTEM

We introduce NetAccel, the first Network-Accelerated MPP database system. NetAccel is a full-fledged MPP database with three novel components that take advantage of programmable switches (see Figure 3): 1) a *Network Scheduler* that identifies appropriate queries, determines the placement of network-accelerated operators on the programmable switch, and monitors their execution; 2) a *Deparser* taking care of the communication between the MPP nodes and the switch; and 3) a set of new *network-accelerated* query operators that are instantiated on the switch and that can be combined in order to execute segments of the query. We present each component below.

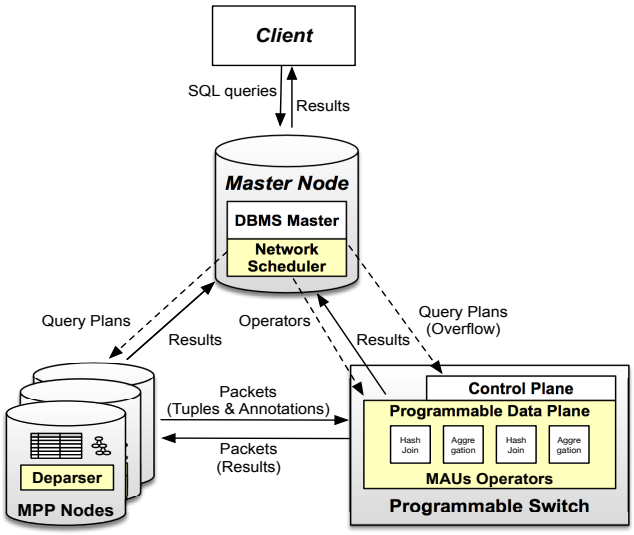

Figure 3: NetAccel's high-level architecture.

3.1 Network Scheduler

The Network Scheduler is responsible for making the dataplane available to queries and for mediating its usage. The first task of the Network Scheduler is to pre-allocate and program a number of MAUs on the switch. The decision of how many and which logic to assign to MAUs is static. Changing that decision requires reprogramming the switch – and some downtime.

One simple strategy to organize the MAUs is to have them implement a fixed query pattern – a join-and-group-by for instance. Ideally, the pattern should be common enough that many queries would benefit. A `P4` program running on the switch would set up a pipeline with the operators that implement the chosen pattern. The program is parameterized over a query instance number and the input relations within it. We call such meta-information about a query its *query context*. The program expects to find the join and group-by column values in an agreed upon position in the packet. At run-time, the Network Scheduler looks for a query instance with a segment that fits the allocated pattern. When it finds one, it informs the switch about the chosen query context. (We discuss how a query sends data to the switch in the following section.) The Network Scheduler then monitors the switch for the end of that query.

An alternative strategy for MAU utilization can be more adaptive. In this scenario, there would be no pattern established *a priori*. A `P4` program running on the switch would set up a number of MAUs independently, and give each of them the ability to execute certain commands, *e.g.*, insert a value into its table, probe a value against its table, increment a value of its table, *etc.* In this scenario, the program would not be parameterized at all. Instead, whenever the Network Scheduler decides to assign one or more MAUs to a query pattern, it would alter the query plan to point to those MAUs. The plan would transmit both the commands and the tuples on which they should act in the same packet. In this sense, one is reminded of adaptive query processing techniques, like Eddies [7].

Irrespective of the MAU allocation strategy, the Network Scheduler has to determine how large the tables in each

MAU should be. That space is used to store an operator's state. The larger the space, the bigger an operator's state can fit. (We will talk about how each operator uses its space in Section 3.3.)

But estimating an operator's state size is notoriously error-prone. Instead of relying on accuracy, we put in place provisions to handle the excess tuples, should they occur. We call this mechanism *overflowing* and task the Network Scheduler with setting it up. One convenient area in which to overflow is the switch local *control plane*, if the switch is an SDN-capable one [18]. Usually, the control plane is itself a general-purpose computer with its own CPU. Alternatively, we can engage one or more MPP nodes to help. Still, we could decide to revert to a non-network-accelerated query plan. Regardless of the chosen strategies, the Network Scheduler puts them in place before the query starts running, as shown in Figure 3. This avoids any coordination overhead to handle overflow at run-time. We discuss overflowing options in more detail in Section 3.4.

3.2 Deparser

The Deparser is the component that manages the communication between an MPP node and the switch. An important consideration here is the choice of a network protocol. For instance, simply making a tuple be the payload of a TCP packet would not work. The switch drops many packets as part of our normal processing, *e.g.*, when building a hash table (see below Section 3.3). Moreover, the switch also creates new packets dynamically, *e.g.*, when forwarding the results from a group-by MAU. TCP being stateful making such changes to the packet flow without a receiver equating them to anomalies would be an overhead.

Better network protocol stacks exist for our case. We could use a traditional IP stack and a connectionless protocol such as UDP. Or use a light protocol directly atop of Ethernet. We are currently exploring the latter, which gives us some advantages. For one, we can design packet headers that are more appropriate for our usage. The Deparser uses the header to convey a tuple's query context. The packet is structured in a way that is straightforward for the switch to parse. For example, if a MAU is executing a join, that column appears in the packet at an expected position.

Another advantage of having a special protocol is performance. At 100 Gbps and assuming tuples of less than 40 bytes, this may mean forwarding more than 148 millions tuples per second per port onto the wire. Currently, normal OS and TCP/IP stacks cannot operate at this pace. We bypass both.

The Deparser interfaces with a query plan via the similarly named `deparse` operator. The operator `deparse` enqueues data for the Deparser component to dispatch. This separation allows all the details of the communication to be ready when data actually needs to be transmitted.

3.3 Network-Accelerated Operators

Implementing an operator requires fitting its algorithm into match-action tables, that is, into a pattern of table lookups and state/packet transformations. A match-action table can only perform a limited number of operations before the packet on which it is acting advances to the next MAU. The more complex an operator's algorithm, the more MAUs are required. Our first design consideration is thus to keep to a reasonable number of MAUs. Another important consideration is the size of the state an operator maintains. As every MAU can only offer a limited amount of memory, we may need to pool several MAUs to implement a given operator. Both these aspects make designing network-accelerated operators challenging.

As we stated before, we believe it is impractical to assume that state size estimates are accurate. *Overflowing* excess tuples may occasionally be necessary. Instead of requiring every operator to handle overflow itself, we make this facility available at the system level. An operator need simply to identify a tuple that overflowed and to keep track that it occurred. (We will show an example shortly.) In the following, we give an overview of some of the operators we have considered and discuss how they were made to fit into a packet-forwarding programming model. We leave the presentation of overflowing techniques to Section 3.4.

Hash-Join: We designed a hash-join algorithm that requires as few as two MAUs – one to store a hash table and one to keep track of overflow. But up to N MAUs can be used to store larger hash tables, if they are available. The Network Scheduler has the flexibility to size the join however it sees fit. We note also that this is an equi-join, and that we assume the outer table's join values are unique.

The algorithm's hash table uses a closed-addressing scheme (collision chains) as we illustrate in Figure 4. Instead of allowing variable-length chains, a fixed length of N is defined. The algorithm preallocates all chains, as the programming model prevents dynamic use of resources. If an insertion finds an occupied slot at MAU k, we let the packet proceed to the next MAU. The algorithm attempts the insertion there, and continues trying until successful or when $k = N$.

If the insertion fails at all the MAUs, the packet should overflow. We emphasize that the algorithm does not need to handle that packet. All it needs to do is reflect in the packet metadata that indeed an insertion was not possible. It should also record that the chain to which the packet hashes is full. We describe the insertion and the probing procedures in detail in Algorithm 1.

Hash-Based Aggregation: As with the join, the aggregation operation can be centered around a hash-table, one that keeps track of unique groups. For illustration purposes, we consider a group-by over a compound column henceforth. (There is one such group-by in TPC-H Query 20. In the process of de-correlating `lineitem`, a group-by over `l_partkey` and `l_suppkey` is introduced.) Normally, a single table can express the logic to find the group to which a tuple belongs and to calculate the aggregation. But if the group itself is compound, we cannot express the extra comparison *and* the aggregation in single MAU.

Our solution is to identify early on if we are inserting a group into an empty slot (skip the comparison) or into a non-empty one (mandatory comparison).[1] We thus initiate the aggregation with a preliminary stage, called *group-by presence*, which identifies when a given hash position is empty. The following stage holds a compound table, and inserts or compares groups accordingly. The aggregation table itself lies in a third stage, indexed by the same hash as the group

[1]This has shown, in practice, to be a common programming pattern. We handle a complex action thanks to a preliminary test that can then choose between two simpler actions.

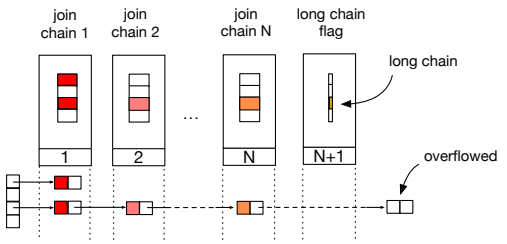

Figure 4: Stage layout for a hash table based on fixed-sized collision chain. The k-th stage, $1 \leq k \leq N$, keeps track of the keys in the k-th position of every collision chain. An extra stage flags the collision chains having more elements than stages. Elements that do not fit are handled in the overflow area.

table. We present this procedure in detail in Algorithm 2.

Data Motion: Data motion operators on the switch go beyond implementing the usual tuple broadcast, redistribute, and gather patterns. After the switch updates an operator's state, the packet that contributed its value is dropped. Thus when the operation is over, the switch must itself produce packets to forward the results downstream. We call the mechanism that moves the results of a completed operation off of the switch *draining*. Data motion operators are built atop of draining, by specifying what distribution pattern to use across the MPP nodes downstream.

Draining starts when the switch processed all input tuples for a running query pattern. Essentially, draining implements an iterator over an operator's results. It reads the first value of the result and puts it into a draining packet. We proceed by resorting to two packet manipulation facilities the switch provides. One is packet cloning. Before sending the draining packet out, we copy it. The second mechanism, packet recirculation, allows us to forward the copy back into the ingress side of the switch. We keep iterating this way until the last result is sent out.

Data Reloading: This is an operator dedicated to relocating data *within* the switch. It is the cornerstone for supporting *composition* over patterns. There is such an opportunity in TPC-H Query 20 for example, when two instances of a join-and-group-by pattern appear. With data reloading, we only need to allocate one of the patterns on the switch at a time. We perform the first join-and-group-by as if it were the sole pattern. We then transfer the state between the group-by stages and the *next* join stages from the second join-and-group-by. The important detail here is that those stages are the same as the ones that the first join used. The join can be reparameterized prior to reloading.

In practice, data reloading uses a similar iterating process as draining. The only addition here is logic that writes data to MAUs with results coming from draining packets. In other words, the results of the previous pattern behave as if they were the outer relation in the next join.

3.4 Overflowing Techniques

We call *overflowing* the moving of part of an ongoing operator's state out of the dataplane. Overflowing may start at any point in query execution. For instance, in our closed-addressed hash table, we may need to insert a new item in a collision chain that is full.

Algorithm 1 Hash-join with Fixed-sized Chains

```
1:  stage 1..N(tbl:int[SIZE])
2:      upon receiving pkt, metadata m do
3:          match pkt.outer ∧ ¬m.found
4:              if tbl[m.hash] = ∅ then
5:                  tbl[m.hash] ← pkt.join_key
6:                  m.found ← true
7:                  mark pkt as dropped
8:              else if tbl[m.hash] = pkt.join_key then
9:                  m.found ← true
10:                 mark pkt as dropped
11:             else
12:                 m.found ← false
13:         match pkt.inner ∧ ¬m.found
14:             m.found ← tbl[m.hash] = pkt.join_key

15: stage N+1(flag:bool[SIZE])
16:     upon receiving pkt, metadata m do
17:         match pkt.outer ∧ ¬m.found
18:             flag[m.hash] ← true
19:         match pkt.inner ∧ ¬m.found
20:             if ¬flag[m.hash] then
21:                 mark pkt as dropped
```

Algorithm 2 Hash-Based Aggregation

```
1:  stage 1(flag:bool[SIZE])
2:      upon receiving pkt, metadata m do
3:          match pkt.inner
4:              m.gby_present ← flag[m.hash]
5:              flag[m.hash] ← true

6:  stage 2(gby:int[SIZE][2])
7:      upon receiving pkt, metadata m do
8:          match pkt.inner
9:              if ¬m.gby_present then
10:                 gby[m.hash][0] ← pkt.gby_val1
11:                 gby[m.hash][1] ← pkt.gby_val2
12:                 m.gby_inserted ← true
13:             else
14:                 if ( gby[m.hash][0] = pkt.gby_val1 ∧
                         gby[m.hash][1] = pkt.gby_val2 ) then
15:                     m.gby_inserted ← true
16:                 else
17:                     m.gby_inserted ← false

18: stage 3(aggr:int[SIZE])
19:     upon receiving pkt, metadata m do
20:         match pkt.inner ∧ m.gby_inserted
21:             aggr[m.hash]+ = pkt.aggr_val
```

A convenient area in which to overflow is the control plane of a switch. The dataplane can reach the control plane simply by routing packets to it.[2] In that sense, the control plane could be seen as any other connected server. The advantage is that the control plane has direct access to all the dataplane state, if needed. The main limitation is the speed of the connection, which is implementation-dependent. Commonly, we expect a single port's worth of bandwidth, sometimes even less.

[2]A similar communication occurs when handling the ARP protocol. When the dataplane *learns* about a new MAC address, it sends it to the control plane which itself adds that address to the appropriate tables.

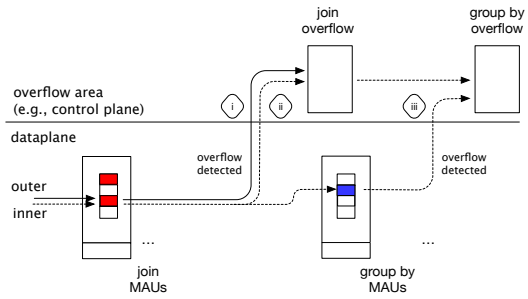

Figure 5: In a pattern of operators such as a join-and-group-by, overflow can happen on the join MAU if a collision chain is full. Tuples from either the outer or the inner relation can overflow, as in (*i*) and (*ii*), respectively. Moreover tuples from the inner relation that did not overflow from the join might overflow at the group-by MAU, as in (*iii*).

Once a tuple is redirected to the control plane, any further operation to that tuple is handled there. This can change the flow of other tuples as well, if they are indirectly associated with the overflowed tuples. For instance, if we offloaded a join-and-group-by pattern onto the switch, tuples that match an overflowed key on the join hash table will also be aggregated by the control plane. We depict this scenario in Figure 5.

Overflowing onto the control plane is efficient as long as there is no queue build up. If the control plane detects a build up, it may opt to handle the overflow outside of the switch also. This would require having the *Network Manager* make that alternative available. It would have done so by installing a variant of the non-accelerated query plan on some MPP nodes and by having the dataplane route overflowed state to those servers. We call such an alternative plan a *fall-forward plan* to emphasize the fact that we do not discard any of the work performed so far. A fall-forward plan is able to handle both the unprocessed tuples the switch might overflow and the results already processed by the switch.

Deciding on when to use and how to rewrite an accelerated query into an efficient *fall-forward plan* is still a work in progress.

4. PROTOTYPE

To evaluate our proposed architecture and the design alternatives we discussed above, we implemented a prototype system to serve as a test-bed for our strategies. We chose to start with the variation where we pre-assign MAUs to a common query pattern, the join-and-group-by one. Our hash-table implementation resorts to fixed-sized collision chains. In addition, we also built some further *overflow* logic such that tuples that fall in longer chains would be routed to the switch's control plane. All our work was performed on an actual programmable switch based on the Tofino silicon [11]. Figure 6 describes the implementation of the query pattern running on the switch down to the MAUs allocation.

For brevity, we do not elaborate on the data motion/reloading here. But we note that we have MAUs to gather, redistribute, or broadcast results to the MPP nodes (or, in the future, to recycle them into a following query pattern without leaving the switch.)

We also made initial steps to implement a Deparser component that injects tuples into the network using a protocol we programmed the switch to recognize. This Deparser bypasses the OS stack by using Intel's DPDK [4], though we are currently working on an RDMA [5] version.

4.1 Preliminary Experiments

Our experiments focus on three key points: (1) the potential gains when offloading query segments onto the switch, (2) the impact of overflowing, and (3) the scalability as the network speed increases.

Our experimental setting is as follows: A programmable switch connects a cluster of 2.1 GHz dual-socket Xeon E5-2620(v4) each with 128GB of memory and a 100 Gbps network card. We run the Greenplum Parallel Database "segment nodes" (*i.e.*, MPP nodes) on three machines and leave a dedicate machine for the "control node" (*i.e.*, Master node). The data used in our experiments was generated by the TPC-H benchmark tooling with a scale factor of 100. We distributed the data uniformly across the MPP nodes. We use `P4` to program the switch's dataplane.

In our first experiment, we compare the effects of running a query with and without network acceleration. We use the most expensive join-and-group-by segment of TPC-H Query 20, over `part` and `lineitem`, as our query pattern. We show the normal and the accelerated query plans used in Figure 7(a) and (b), respectively, including the response times obtained running our network at 25 Gbps. To isolate any disturbing factor, both versions of the query start with the same data content, but formatted differently. Recall that in Query 20 the `part` relation is filtered over `p_name` and projected over `p_partkey`. The `lineitem` relation is filtered over `l_shipdate` and projected over `l_partkey, l_suppkey, l_quantity`. We created relations `part_FP` and `lineitem_FP` – Filtered and Projected – to store the initial data in Greenplum. These are used by the non network-accelerated plan. For the accelerated one, we created `part_FP'` and `lineitem_FP'` in a format our `deparse` operator can read. The pairs of relations are distributed in the exact same way, over `l_partkey, l_suppkey`.[3]

In the normal plan, Greenplum tries to move as little data as possible and to perform the join-and-group-by locally. This is the accepted best practice for distributed plans [24]. Conversely, the accelerated plan pushes *both* relations onto the switch. The effective network speed the `deparse` achieved in that case was within 2% of the nominal maximum of 25 Gbps. The accelerated plan ran 2.04x faster.

In our second experiment, we wanted to quantify the overflow in the previous experiment's query and determine its impact on response time. Note that the overflow can occur in three different cases, as depicted in Figure 5. The number of rows overflowed heavily depends on the distribution of stages among the operators of a given pattern. We started our experiments using two (main) stages for the join and two for the group-by. If, instead, we had a total of six stages to use between them, we would have the overflow scenarios described in Table 1. That table shows that, in terms of sheer amount of overflow, it is more advantageous to assign extra stages to the join rather than to the group-by. As this join is

[3]Because we are now distributing `lineitem_FP` over the same columns as the group-by, there is no need to split that operation into a local and global operators. This is to the advantage of the normal, non network-accelerated plan.

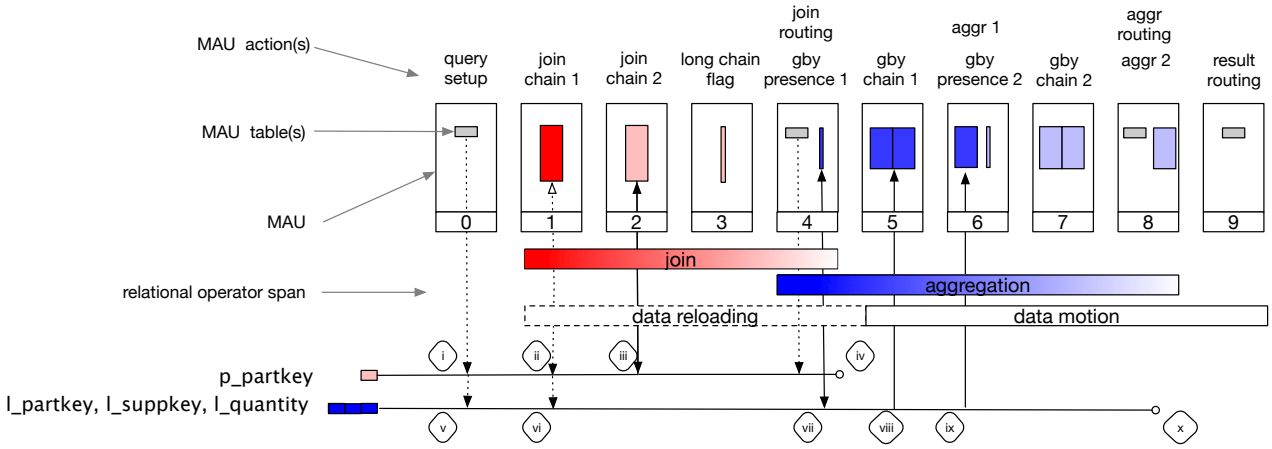

Figure 6: The join/group-by/reshuffle logic, as compiled from P4 sources, takes 10 MAUs. We are using hash-tables with fixed collision chains of two elements for both the join and the group-by. (Single column for the join and double for the group by.) Incoming tuples from the MPP nodes are processed at (*i*) and (*v*) where a tuple's join and group-by hash indices are computed. The outer relation tuple (in red) reads the first entry in the collision chain (*ii*). That entry is occupied, so the tuple moves on to the next stage and gets inserted as a collision (*iii*). The tuple then gets marked to be dropped in (*iv*). The inner relation tuple (in blue) first probes the join hash-table. It matches a key in the first entry of the collision chain (*vi*) and so is allowed to proceed down the pipeline. In (*vii*), it identifies that its corresponding group-by table index is empty. It sets a bitmap to indicate it will take the position and gets inserted into the group-by hash table in (*viii*). It moves to the next stage where the initial aggregation value for that group is recorded (*ix*) before the tuple is marked to be dropped at (*x*).

very selective, the more join state we keep on the dataplane, the more tuples can be dropped there that failed the join.

We further broke down the number of rows that overflowed because of each of the cases described in Figure 5. The results are shown in Table 2. As expected, giving more stages to the join reduces the overflow at the join stage. More rows were then processed on the dataplane. As a result, the overflow of the group-by increased, but by a negligible amount. Not surprisingly, the decision on how to distribute the available stages was a cost-based one.

To evaluate the overflow numbers, one should consider: (a) how large of a connection between the dataplane and control plane is needed to transfer tuples without queuing; and (b) how powerful a CPU is required to process the overflow, also without queuing. In our experiments, if we assign 4 or 5 stages to the join, the bandwidth necessary to overflow is largely within that of a 10 Gbps card and within a servers-class server's reach. Our current programmable switch control plane has the ability to process it using a single thread. In contrast, the more stages we give to the group-by, the more bandwidth and CPU power is required to not slow down the computation on the switch.

In the third experiment, we investigate the effects of varying the network speed. The results are shown in Figure 8. We use the full TPC-H Query 20 in four different plans: the plan originally chosen by Greenplum ("orig"), the agreed upon best plan of the query [24] ("normal"), an offload simulation where all the joins/group-by's would be performed on the switch without overflow queuing ("accel"), and lastly the lower-bound response time of the network-accelerated plan achievable if one systematically saturated the network ("min").

We observe that the results for the original and normal plans vary very little. This is not surprising, as they were designed to minimize network traffic. Still, the performance discrepancy between the original plan and the normal plan is considerable. That's because the original plan starts by

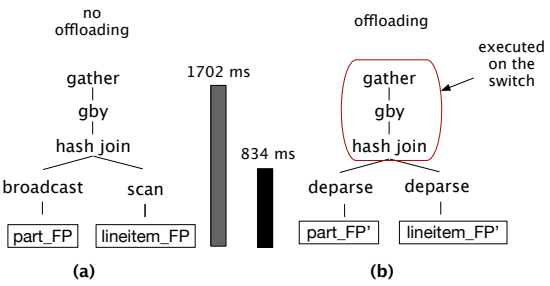

Figure 7: The (a) normal and (b) accelerated plans for the first join/group-by pattern of TPC-H Query 20. The bars indicate response times.

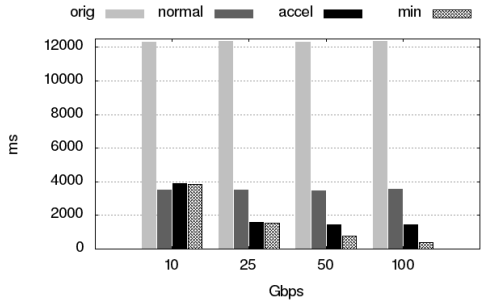

Figure 8: Effects of varying the network speed on four plans for TPC-H Query 20. Accelerated plans respond better to speed increases.

# of join stages	# of group-by stages 1	2	3	4	5
1	49.5%	49.3%	49.2%	49.2%	49.1%
2	23.5%	23.4%	23.3%	23.2%	
3	9.33%	9.21%	9.09%		
4	3.58%	3.46%			
5	1.62%				

Table 1: Percentage of overall number of tuples overflowed as a result of varying the number of stages assigned to the join and the group by. Assigning more stages to the group by, in this scenario, increases the number of tuples overflowed. We show the breakdown of the five cases in gray in Table 2

.

# of stages join x group-by	overflow points (*i*)	(*ii*)	(*iii*)
1x5	51.1%	49.0%	0.07%
2x4	21.3%	22.7%	0.38%
3x3	7.36%	8.45%	0.65%
4x2	2.16%	2.64%	0.82%
5x1	0.52%	0.67%	0.96%

Table 2: Breakdown of the overflow when we assign n x m stages to the join and group-by, respectively. The label (*i*) refers to percentage of overflowed tuples of the outer relation; (*ii*) and (*iii*) are inner relation percentages, at join or at group-by, respectively (cf. Figure 5).

joining `part` with `partsupp`, instead of the much more selective join between `part` and `lineitem`. Another difference – likely related to the choice above – is that we suspect that Greenplum misses an essential transformation the normal plan uses. It does not push a join across a group-by. (Whereas it does the other way around.)

In turn, the plan variants that engage the switch see a linear speed up from 10 to 25 Gbps. Past that point, the accelerated scenario gains little of the potential speedup because the current version of our `deparse` operator plateaus at 29 Gbps. Realizing the potential speed – the difference between the "min" bar at a given speed and the "accel" one – requires some future work (*e.g.*, by using RDMA).

5. DISCUSSION

Despite the potential performance gains, the adoption of network-acceleration in query execution raises a number of questions. The first one concerns the cost of such a special switch. The complexity added to the chip area of a programmable MAU versus a *fixed-function* equivalent unit was determined to be 14.2%; the additional power requirement, 12.4% [11]. Thus in terms of cost of acquisition and operation, the technology is competitive.

The second common question refers to usability. Once some MAUs are assigned to query execution, how much of the dataplane is left for actual packet forwarding? There's no question that table area given to the database competes with routing needs. But the query in our running example, for instance, never reached above 60% of a MAU's resources in terms of SRAM usage. And there were several MAUs in which our usage was under 20%. It seems that the share of the switch that we took could be compatible with the usage a *top-of-rack* switch would require. Then again, depending on the fabric of a datacenter, TOR switches can have very large L2-related tables. Assigning switch resources to an application should then be planned in conjunction with a datacenter interconnect.

A third question is about the network efficiency that our techniques achieve. The sacrifice we made was to use as short a packet as needed to hold a single tuple, as `P4` does not naturally support iterating over an array of objects inside a packet. The Ethernet frame header represents a significant overhead in such small packets. Even if there are ways to mitigate the overhead (e.g., by re-purposing header fields), we cannot eliminate it altogether.

The fourth question is about security. There is currently no support on the switch for it to participate in secure communication channels. The data the switch manipulates cannot be encrypted. There are, however, mechanisms that would prevent such data from being tampered with. There are also ways to prevent the queries from being altered or sent by unauthorized users. The point here is that even if privacy can be difficult to handle, other aspects of security such as authentication and integrity can be supported.

6. RESEARCH AGENDA

We believe NetAccel serves as a foundation for future systems that can fully incorporate programmable switches into MPP systems. This work opens several fundamental research directions that need to be explored, including:

CPU Offloading: Executing full query segments on the switch allows us to save CPU on the MPP nodes. This however assumes that the extra work performed to move tuples onto the network does not offset the gains. *Deparsing* tuples onto the network without CPU assistance is then critical. We are currently exploring techniques such as RDMA for deparsing. Another alternative we are considering is using smart NICs to aid in the deparsing.

Data Overflowing: The control plane overflow techniques we described are but a first step. We can further exploit the control plane's seamless access to the dataplane. When redistributing the results of a query, for example, the control plane can move its share back onto the dataplane. This may be possible because, as some stages finish their role in a given query, their tables get freed. Draining could then be done from the dataplane only, rather than needing to include the overflow area. We also mentioned *fall-forward plans*, which could present a second degree of overflowing after the control plane. Finding plans to join the final and intermediate results on the switch with unprocessed rows should be possible and efficient. Then again, as query execution speeds increase, the extra result-gathering overhead might hurt the data processing effort itself. Lastly, we suspect that other variations of well-known algorithms we describe here (*e.g.*, streaming) would need less and less state on the switch. The long-term vision is to co-design both operators and data structures with strict space constraints and overflow in mind.

Efficient and Flexible MAU allocation: We would like to identify other common query patterns than join-and-group-by's that could be preloaded on the switch. Obvious choices are different combinations of these operations, *e.g.,*

join-and-join, group-by-and-join-and-group-by, *etc.* We are looking for representative workloads from which to uncover such patterns. We are also considering more dynamic use of the MAUs by carrying information on how to process a tuple in its packet directly, rather than preloading a query pattern on the switch.

Switch Parallelism: One facet of fast switches we have not explored thus far is the fact that they can have several parallel pipeline instances. A switch with 64 ports can actually be composed of four distinct groups of 16 ports that share nothing but a traffic manager. (The equipment that we are experimenting with is built that way.) As we scale the number of MPP nodes, distributing the work across several pipelines becomes unavoidable. We could also scale the number of MPP nodes via a fabric of switches. These questions open at least two areas of future research: parallelization of network-accelerated operators and network topologies for network-accelerated systems.

Query Optimization: Both the query optimizer and its cost model need to be rethought in our context. Traditional MPP query optimization aims to minimize data movement. But faster network-accelerated strategies come from minimizing the size of intermediate states in a plan. We suspect that these two goals can be conflicting. We know of at least one plan for TPC-H Query 20 that moves more data than the normal one, and yet uses much smaller intermediate states and executes faster on the switch. We also note that it would be useful to explore query plans with predictable intermediate sizes. Our goal is to develop a systematic approach to reason about such trade-offs and to equip a modern query optimizer with the means to enumerate and select *optimized network-accelerated query plans*.

7. RELATED WORK

To the best of our knowledge, this is the first work to offload full SQL query segments onto a programmable dataplane. But other examples of *in-network* data processing exist. *In-band Telemetry*, the closer body of work to ours, analyzes network transmission metadata – *e.g.*, queue sizes on the switches or packet transmission delay averages – generated by the programmable switches. Queries in telemetry aim to detect problems (such as network congestion) fast enough to act before the problem propagates.

In-network aggregation for telemetry data was discussed in [21]. A programmable switch acts as a cache for ongoing aggregations. If the switch tables fill up, a group entry is evicted and sent to a full-fledged *backing store* charged with merging that group's intermediate results. In contrast, our *overflowing* technique partitions the work across the dataplane and other overflow areas.

Exploring query plans that aggregate data while using less state is discussed in [14]. Their *dynamic query refining* takes advantage of the hierarchical properties of some attributes, such as an IP address (i.e., hierarchy of sub-nets). This aggregation can be done at incremental levels of granularity using the attribute's hierarchy.

The scenarios which an MPP database and a telemetry application are designed for are different, though. Network monitoring involves but a small fraction of all data being transmitted, its schema opportunities are known at switch compile time, and the data is generated by the switches themselves. In contrast, MPP databases may offload any portion of the data onto the switch, their schemas may not be known until after the switch is deployed, and data is generated by MPP nodes.

Our work joins an ongoing effort of revisiting how the network and database systems interact. One such approach appeared in [28]. The authors suggest how to use SDN – the control plane of the network – to obtain information relevant to the query optimization process. We complement this approach by making the dataplane of SDN networks an active portion of the query execution engine, whenever programmability is available. Another approach involves leveraging the network speed increases. An architectural redesign of databases in response to that appears in [8, 23]. With the advent of RDMA, the authors argue that networking is no longer the bottleneck in distributed query execution. We agree. The authors then describe distributed algorithms, for instance for joins, that take better advantage of the network. Here too, our approach is complementary, as operating with RDMA packets on the dataplane is viable. Lastly, other frameworks than SQL have been considered for acceleration as well. Map-reduce based ones are an example [9]. These frameworks target a fabric of switches rather than individual instances.

Programmable switches have also been used in aspects of data management beyond query execution. There are a number of techniques for in-network support for concurrency. A resource coordination service based on in-network access requests is presented in [16]. By moving the processing of such requests to the switch, the number of round trips a resource acquisition protocol uses can be vastly reduced. The protocol being lightweight, this allows for lower contention scenarios. A technique to serialize sharded transactions via an in-network sequencer is presented in [19]. By virtue of listening to all transaction requests at line speed, a serial order for a class of *one-shot* transactions can be determined without any coordination between the participants. Having database operations beyond query execution be executed in-network can only increase the chances that future commercial databases would be designed with programmable network devices in mind.

8. CONCLUSIONS

In this paper, we introduced NetAccel, the first network-accelerated DBMS. Our system takes advantage of new physical operators to run entire segments of a query plan at line-speed on a programmable switch. While realizing the full potential of our vision will take years, we are excited by the prospect of NetAccel and by the new data processing avenues opening up with the advent of next-generation networking equipment.

ACKNOWLEDGMENTS

We would like to thank Robert Soulé, Huynhtu Dang, and Theodore Jepsen for some early discussions on the Tofino architecture and setup, Feng Tian for numerous clarifications about Greenplum, Niall Dalton and Evan Jones for valuable comments on early drafts of this paper, and André Ryser for helping in the CPU overflowing experiments.

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement 683253/GraphInt).

REFERENCES

[1] Barefoot: The world's fastest and most programmable networks. https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[2] Cavium xpliant ethernet switch product family. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[3] Cisco quantumflow processor. https://newsroom.cisco.com/feature-content?type=webcontent&articleId=4237516.

[4] Data plane development kit. https://www.dpdk.org.

[5] Rdma and roce for ethernet network efficiency performance. http://www.mellanox.com/page/products_dyn?product_family=79.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. NSDI. USENIX Association, 2010.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2), 2000.

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7), 2016.

[9] M. Blöcher, T. Ziegler, C. Binnig, and P. Eugster. Boosting scalable data analytics with modern programmable networks. DAMON, 2018.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), 2014.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. SIGCOMM, 2013.

[12] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. SIGCOMM, 2017.

[13] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, 2013.

[14] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. SIGCOMM, 2018.

[15] P. Gupta and N. McKeown. Algorithms for packet classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2), Mar. 2001.

[16] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. NSDI, 2018.

[17] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. NSDI. USENIX Association, 2015.

[18] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 2015.

[19] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. SOSP, 2017.

[20] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[21] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. SIGCOMM, 2017.

[22] R. Ozdag. Intel ethernet switch fm6000 series - software defined networking. *https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf*.

[23] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1), 2017.

[24] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query optimization in microsoft sql server pdw. SIGMOD, 2012.

[25] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. SIGCOMM, 2016.

[26] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. *SIGCOMM Comput. Commun. Rev.*, 1997.

[27] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander. Rpl: Ipv6 routing protocol for low-power and lossy networks. Technical report, 2012.

[28] P. Xiong, H. Hacigumus, and J. F. Naughton. A software-defined networking based approach for performance management of analytical queries on distributed data stores. SIGMOD, 2014.

[29] F. Zane, G. Narlikar, and A. Basu. Coolcams: power-efficient tcams for forwarding engines. INFOCOM, 2003.