AQUOMAN: An Analytic-Query Offloading Machine

Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim*, Sungjin Lee*, Arvind
*MIT CSAIL, DGIST**
{shuotao, bthom, tianhaoh, arvind}@csail.mit.edu, {ghwns9652, sungjin.lee}@dgist.ac.kr*

***Abstract*—Analytic workloads on terabyte data-sets are often run in the cloud, where application and storage servers are separate and connected via network. In order to saturate the storage bandwidth and to hide the long storage latency, such a solution requires an expensive server cluster with sufficient aggregate DRAM capacity and hardware threads. An alternative solution is to push the query computation into storage servers.**

In this paper we present an in-storage Analytics QUery Offloading MAchiNe (AQUOMAN) to "offload" most SQL operators, including multi-way joins, to SSDs. AQUOMAN executes *Table Task*s, which apply a static dataflow graph of SQL operators to relational tables to produce an output table. Table Tasks use a streaming computation model, which allows AQUOMAN to process queries with a reasonable amount of DRAM for intermediate results. AQUOMAN is a general analytic query processor, which can be integrated in the database software stack transparently. We have built a prototype of AQUOMAN in FPGAs, and using TPC-H benchmarks on 1TB data sets, shown that a single instance of 1TB AQUOMAN disk, on average, can free up 70% CPU cycles and reduce DRAM usage by 60%. One way to visualize this saving is to think that if we run queries sequentially and ignore inter-query page cache reuse, MonetDB running on a 4-core, 16GB-DRAM machine with AQUOMAN augmented SSDs performs, on average, as well as a MonetDB running on a 32-core, 128GB-DRAM machine with standard SSDs.

***Index Terms*—Accelerator; SQL analytics; Near-data computing; FPGA; Flash storage; Database**

I. Introduction

Multi-terabyte/petabyte datasets are now commonplace for analytic workloads. In 2017, Uber generated 100TB of trip-tables daily, and analysed 100PB of data for business intelligence [5]. In many cases, the data is stored in relational format on hard drives and analyzed by SQL database software such as Presto [2], Vertical [1], and MonetDB [4]. To process an analytic query, the database software brings the input data on demand from hard drives to DRAM, and then uses powerful CPUs to compute with this data. In a data warehousing architecture in the cloud, application servers and storage servers are separate and connected via a network [6]. Application servers initiate analytical queries, fetch data from the central storage and then process it. Such a "disaggregated" architecture is popular in the cloud because customers can scale the application servers and storage servers independently. For fast query responses, analytical software typically requires application servers to have sufficient hardware threads (i.e., virtual cores) and DRAM to hold the input data to overcome the long latency, large access granularity and limited bandwidth of central storage accesses. In April 2020 the largest storage-optimized Amazon EC2 server (i3.metal) can accommodate 8 1.9TB SSDs, and is equipped with 72 virtual cores and 512GB DRAM. Such large processing power and DRAM are needed to be able to fully exploit the high-bandwidth of SSDs. Storage throughput, because of advances in flash technology, has improved by 13X in the past decade [42], and has greatly outpaced CPUs ability to process data in memory [20, 37]. As denser and faster storage devices become available in the future, it will become increasingly difficult for storage servers to provide sufficient CPUs and DRAM to have a cost-effective balanced system.

An alternative solution is to push part of query processing to the storage to eliminate unnecessary data movement. Such a solution has been deployed in several commercial systems, for example, Oracle Exadata Server [22], IBM Netezza Machine [46], and IDM [48]. One of the most recent systems is Amazon Web Services (AWS)'s "S3 Select" feature, which pushes filter operation to the shared cloud storage service, and can get up to 4X performance benefits for these operations [3]. In this paper we propose an in-storage Analytic QUery Offloading MAchiNe (AQUOMAN), which pushes this idea of "off-loading" query processing to storage much more aggressively.

AQUOMAN's programming model is based on a sequence of *Table Task*s, each of which applies a static dataflow graph of SQL operators on an input table in a streaming manner to produce an output table. It takes inputs from flash, in a file format used by column-oriented database like MonetDB, because it is better suited for analytic workloads. Given the SQL query execution plan - a tree of SQL operators - we identify the subtrees that can be directly translated into *Table Task*s. By employing a streaming model, AQUOMAN significantly reduces the DRAM requirement for intermediate tables.

We want to keep the memory in AQUOMAN to be small enough, say 16GB per 1TB-SSD, so that AQUOMAN can be *embedded* in an SSD. This prevents us from fully off-loading some queries, for example, a *multi-way join*, whose intermediate tables exceed the DRAM capacity of AQUOMAN. Despite these limitations (Sec. VI-E), AQUOMAN can profitably execute the majority of queries in the TPC-H benchmark suite on a 1TB dataset, giving us an opportunity to reduce both number of hardware threads and DRAM usage in the host. We will show, using TPC-H benchmarks on 1TB data sets, that a single instance of 1TB AQUOMAN disk, on average, can free up 70% CPU cycles and reduce DRAM usage by 60%. Thus,

Accel. Type	Related Work	Impl.	Supported SQL operator	Evaluated TPC-H Queries	Data Sz.(GB)
In-memory	Q100 [52, 53],MasterOfNone [35]	ASIC	All	All 22 queries w/o regular expression	0.01
In-storage	SmartSSD [18]	ARMs	*Filter*, *Aggregate Group-By*	Q6, 2 custom queries	100
	Summarizer [33]		*Filter*	Q1,6,14, a custom query	0.1
	Biscuit [23], YourSQL [28]	ARMs/ASIC	*Filter*	All (8 is partially offloaded)	100
	Ibex [51]	FPGA	*Filter*, *Aggregate Group-By*	Q13, 6 custom queries	10
	Insider [42]		*Filter*	A Custom Query	60
	FCAccel [50]		*Filter*, *Aggregate Group-By*, *Arithmetic*	Q1,6, and a custom query	100
	AQUOMAN	**FPGA**	**All**	**All** (14 are fully offloaded)	**1000**

TABLE I: Representative near-data SQL accelerators

replacing standard SSDs with AQUOMAN SSDs in database systems is a sound economic proposition.

One way to visualize this saving is to think that if we run queries sequentially and ignore inter-query page cache reuse, MonetDB running on a 4-core, 16GB-DRAM machine with AQUOMAN augmented SSDs performs, on average, as well as a MonetDB running on a 32-core, 128GB-DRAM machine with standard SSDs.

We make the following contributions in this paper:

1. AQUOMAN, a novel microarchitecture for an in-storage accelerator capable of stream processing a *Table Task* which is a static dataflow graph of operators;
2. A complete in-storage solution which can be fully integrated into a database management software (DBMS);
3. An FPGA-based implementation of AQUOMAN, which can process data stream at the line-rate of our flash controller;
4. An end-to-end evaluation of AQUOMAN using TPC-H benchmarks on 1TB data-set against the baseline of an x86 server with 16 dual-threaded cores and 128GB DRAM.

Paper Organization: We begin with related work in Section II, followed by examples of translating SQL queries into dataflow maps (Section III). We then give an overview of AQUOMAN (Section IV) and its programming model (Section V). It is followed by the detailed microarchitecture (Section VI) and implementation of AQUOMAN (Section VII). We evaluate the performance of AQUOMAN in Section VIII, followed by a brief conclusion (Section IX).

II. Related Work

There is a long history of attempts to use specialized hardware to accelerate database query processing [9, 16, 17] but it has never caught on. One reason is that the dramatic increase in processing power and DRAM capacity of commodity hardware over the last four decades has reduced the incentive to use special hardware. However, with the rise of specialized hardware in datacenters [11, 29, 41, 44] and the increase of storage throughput in the last decade [42], there is a resurgence of interest in accelerating *analytical workload* using FPGAs or ASICs.

Table I gives a summary of the recent work to accelerate database operations near storage. The first family of In-Storage Processing (ISP) architectures leverages the existing ARM cores of the SSD controller to offload simple tasks like filtering [18, 33]. However, the embedded cores in the SSD controller can be 100X slower than x86 cores [47], and offloaded programs can suffer 10X slowdowns [13]. Biscuit [23] and YourSQL [28] use embedded processors in conjunction with a pattern-matching ASIC to offload filtering. They showed offloading filtering is profitable only when the selectivity is sufficiently high.

Another class of ISP (e.g. Insider [42]) uses FPGAs to add processing power to SSDs for a variety of applications to saturate large internal disk bandwidths. One example application of Insider is offloading database filtering, and it provides performance benefits similar to the one provided by a high-end ARM-based solution. Ibex [51] and FCAccel [50] use FPGAs to offload more SQL operators, such as *Aggregate Group-By*, but do not provide a plan to offload a join, which is one of the most dominant operators in analytic queries such as TPC-H.

Unlike existing ISP approaches, AQUOMAN offloads all major types of SQL operators in storage, including the computation-intensive join. AQUOMAN, given an SQL query plan, regroups SQL operators as *Table Task*s, which is the programming model for AQUOMAN's streaming architecture. This enables a transparent integration of ISP with the existing DBMS software. It is also important to note that previous database accelerator research has used much smaller data sets (10MB to 100GB [14, 18, 23, 28, 33, 50, 52]) for evaluation (Table I) and has not addressed the issue of computing with large dataset. In the rest of this section we provide a more detailed discussion of the related work.

In-storage big data analytics framework: As early as 1980's, researchers looked for methods to push computation down into mass storage to process terabytes or even petabytes of data [17]. Following are some of the in-storage frameworks that have been proposed: DBMS [18, 28, 50], graph analytics [31, 34, 36], HPC applications [47] or general workloads [13, 15, 23, 26, 30, 33, 38, 42, 43].

The main difference between databases and other big data applications is that the later usually requires running complex programs on large data structures designed for the purpose, while databases are more specific and focus on running structured queries on relational tables. The shared concerns include how to reduce DRAM requirements, reduce network traffic, and exploit the massive internal flash bandwidth.

General database accelerators: Q100 [52, 53] and its newer variant [35] are general query accelerators based on a programmable spatial-array architecture. Both systems assume inputs and outputs are consumed and produced in the main memory. In terms of executing SQL operators in a data-flow style, AQUOMAN is similar to Q100 and its variant but it addresses the main bottlenecks that both architectures ignored: 1. *Scalability to larger dataset:* Q100's speedup over single-thread software dropped 10X-100X on 1GB TPC-H, and it disappeared almost completely in comparison to multi-threaded software [52]. The functional tiles for sort and join in Q100's ASIC prototype can handle up to 1024 records at 315MHz

on a 256-bit datapath (10.08GB/s) [52, 53]. This forces Q100 to divide-and-conquer large input tables to a huge number of small partitions which causes poor scalability. We have drastically improved the sorter and the join functional units in AQUOMAN. Our FPGA sorter can stream-sort *1GB data* at 12GB/s, and *256GB data* at 6GB/s if there is enough DRAM accessible to the sorter. We ran AQUOMAN on a 1TB TPC-H and still showed speedup over a 32-thread software baseline.

2. *Routing between functional tiles:* Q100 architecture [53] is built around a complex 2D-mesh network-on-chip (NOC), which takes 30-50% of the area and could be challenging to implement in practice. In their more recent work [35], instead of establishing arbitrary connections between heterogeneous tiles, they chose a fixed grid of homogeneous core. The routing simpler but now each tile needed the capabilities of all the heterogeneous cores of [53]. If the tiles are designed to process big workloads, its size will become too big to be realistic. AQUOMAN addresses this issue using a hybrid solution, which supports the common dataflow with a fixed pipeline of three different programmable units: selection, map, SQL swissknife(join/sort/aggregate).

Oracle's RAPID [7, 8] has a rack-scale many-core system specialized for big data analytics. At its core sits a power-efficient general-purpose processor aided by hardware acceleration for data movement and data partitioning. Unlike AQUOMAN's streaming model, the execution model of RAPID is essentially running map-reduce on many cores. Only very primitive SQL operators, bit-vector load and filter, are hardware-accelerated and exposed as special CPU instructions. Mondrian Database Engine [19] employs a similar approach but uses general-purpose cores with SIMD extension as a near-memory processor (NMP) on the logic-layer of a stacked Hybrid-Memory Cube (HMC).

Accelerators for certain database operators: Examples of research focused on implementing specific database operators in hardware include: selection [10, 28, 51, 54], hash join [24, 32], sort-merge join [10, 12], group-by aggregation [51], pattern matching [40, 45], and table histogram generation [27]. Most of these accelerators are attached to memory while a few operate in storage [27, 28, 51]. Operator-specific accelerators assist host-side query execution by task offloading. Our work may use similar operator implementation but our focus is on entire query execution in storage.

FCAccel [50] aggregates SQL accelerators for selection, data aggregation, and hashing on a PCIe-attached FPGA. Like AQUOMAN, FCAccel allows stream processing of selection and aggregation, but used a different technique by dividing tables into small data segments buffered in DRAM. FCAccel is reported to have similar performance as MonetDB running on RAM-disk. FCAccel proposes a collaborative solution with the DBMS software for two-way join. Using a custom query FCAccel shows two tables can be filtered and pre-aggregated and later hash-joined by the host opportunistically. Unlike AQUOMAN, it does not offer a plan to offload multi-way joins.

Query-specific reconfigurable accelerators: To avoid the complexity of designing a general query accelerator, some researchers propose to reconfigure FPGAs for a specific query. SQL operators are implemented as hardware libraries in advance, and are then called and assembled for a particular analytical workload. For example, [14, 49, 55] provide flexible hardware templates for common database operators, while [35] proposes a CGRA architecture where reconfigurable tiles are organized in a systolic manner. The cost of this methodology comes from both the reconfiguration overhead as well as the requirement of using reconfigurable hardware. Baidu [39] has a hybrid solution; certain fixed-function tiles are connected by default in a way that is similar to Q100, but some tiles are reconfigured on demand.

III. Dataflow map of a query

We will first discuss the anatomy of query processing on tables and then describe how these steps are mapped on AQUOMAN.

Single table query: First, consider the query over a single table `sales_transactions` shown in Fig. 1.

```
SELECT
 department,
 sum(price*(1-discount)) \
  as netsale,
 sum(price*(1-discount)*\
  (1+tax)) as revenue,
FROM sales_transactions
WHERE saledate<='2018-12-01'
GROUP BY department;
```

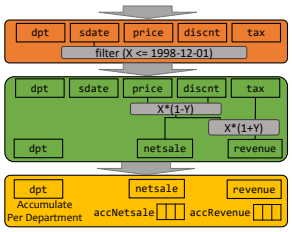

Fig. 1: Dataflow of an Aggregate Query

The columns of the `sales_transactions` table are <`tr-ansactionID, department, saledate, price, disc-ount, tax`>. Each row corresponds to a purchase identified by a unique `transactionID`, which is the primary key for this table and the cheapest way to refer to its rows. From a semantics perspective, the query of Fig. 1 should return the net sale and revenue of each department before 2018-12-01. To produce such an answer, the DBMS typically makes what is called a query plan, for example:

1. Filter all rows of the table verifying a predicate: here the `saledate` value should be smaller than 2018-12-01.
2. Produce a new intermediate table of three columns <`depa-rtment, netsale, revenue`>. Each row of this intermediate table is computed purely from each row selected in Step 1. The department value is directly reported from the incoming row, while the net sale and revenue values are simple arithmetic computation based on the price, the tax and the discount value of the input row.
3. Produce the output table by aggregating the data in the intermediate table, grouped by department.

Note that the first two operations are *map*: they apply functions on each row independently. The last step aggregates data coming from different rows. Those 3 steps can be thought of as a dataflow graph which define how rows of the input table contribute to the query's answer (See Fig. 1). Actually, the

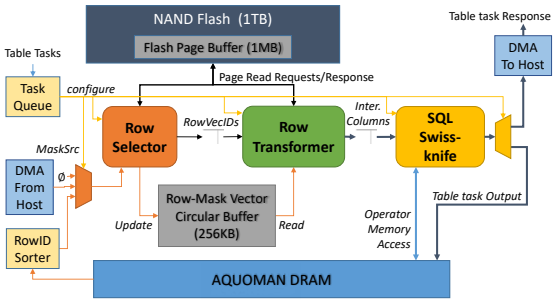

Fig. 2: Overall Architecture of AQUOMAN

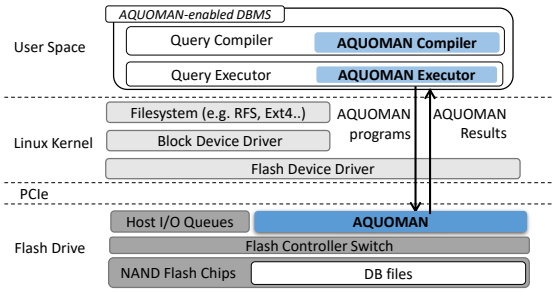

Fig. 3: System Stack with AQUOMAN

```
SELECT
 sum(price) as shoe_sales
FROM inventory as ti,
 sales_transactions as ts
WHERE
 ti.invtID=ts.invtID
 and ti.category="Shoes"
 and ts.saledate>'2018-3-15';
```

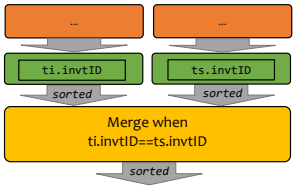

Fig. 4: Dataflow of a join query

dataflow of this particular query illustrates a common plan of row filtering, intermediate table generation and final reductions. Of course the functions used for row filtering, table creation and reductions are query specific.The commonality and high value of this fixed but parameterized dataflow in analytics query processing makes it possible to design a fixed accelerator to process such queries efficiently.

***Join - a multiple table query**:* Suppose the `sales_transactions` table has an extra column to indicate the purchased item. The purchased item is represented as the `inventoryID`, which is the primary key of another table, the inventory table. The `inventory` table has many columns: `<inventoryID, category, quantity, productname, ...>`, where category represents the type of an item.

The following query (Fig. 4) computes the total sale of `shoes` sold after "2018-03-15".

This query needs to compute a so-called inner equi-*join* on the tables `inventory` and `sales_transactions`. Typically, this join query would be processed as follows:

1. Select all the rows that have category "shoes" in the `inventory`.
2. Produce an intermediate table `<transactionID, inve-ntoryID>` from the `sales_transactions` table; to get all the items (referred to by `inventoryID`) that were sold after 2018-03-15.
3. Merge the two intermediate tables produced by the two previous step: every items that are shoes (intermediate table out of Step1) is filtered based on if there exists a sale of that item within the date specified (intermediate table from Step 2).

Notice that the Step 1 and 2 are similar to the steps in the previous example; however in this example they are working on two different tables. In contrast, Step 3 does not seem to fit into the fixed dataflow illustrated in the previous example: it merges data from two intermediate tables.

If we assume that the intermediate tables are generated sequentially and stored in the accelerator DRAM, then a streaming sorter can be placed between the producer and the DRAM, to make the joining easy. Typically these intermediate data (the keys involved in the join) are small enough to be stored in few Gigabytes for Terabytes datasets. In rare cases where the intermediate tables is bigger than the accelerator DRAM, AQUOMAN would relinquish processing to the host. Joins require fast *hardware sorters* to keep up with the streaming rate of the underlying storage. Two-way join generalizes to multi-way join by iteratively storing the sorted intermediate tables in the accelerator DRAM.

IV. Overview of AQUOMAN Architecture

AQUOMAN targets accelerating column-oriented databases like MonetDB. We chose MonetDB because in our TPC-H benchmark evaluation, on average MonetDB was 2X faster than a commercial row-oriented database. In column-oriented database systems, a relational table is stored as a collection of column files. Each column file stores a sequence of column values in ascending row order in either compressed or uncompressed format.

A modern flash drive has huge I/O bandwidth which can easily produce more than one column value per "data beat". For example, a flash hardware controller running at 125MHz with 4GB/s bandwidth is able to produce 32 bytes - equivalent to 8 32-bit column values - per clock cycle.

To allow line-rate data processing, AQUOMAN processes the column data files as a collection of *Row Vector*s, which consists of 32 column values of consecutive rows, indexed by *Row-Vector ID*. A bit-vector that marks which rows have been selected for processing is also stored as part of the table. The overall architecture of AQUOMAN is shown in Figure 2.

The heart of AQUOMAN consists of 3 accelerators *Row Selector*, *Row Transformer* and a *SQL Swissknife*, corresponding to accelerators for the three kind of dataflow operators identified in the previous section. AQUOMAN also relies on one extra *Sorter* to keep the intermediate streams ordered on the required keys.

The *Row Selector* generates the bitvector masks used to efficiently select the input table data (see Section VI for more details on the expressivity of the *Row Selector*). The columns of the rows that have not been masked and are necessary to compute the intermediate table are then streamed to the *Row*

Transformer. The *Row Transformer* is composed of a collection of Processing Elements organized to apply a stateless function on each row to produce a new intermediate table. Finally, the generated rows are fed into the *SQL Swissknife*. The *SQL Swissknife* contains accelerators to perform the standard SQL operators: accumulate, sort, merge, computes the biggest k values ...

The *SQL Swissknife* is equipped with a direct access to AQUOMAN's DRAM, it can leave an intermediate reduced table in it, or consume an intermediate table from it. We will see the usefulness of that patterns when discussing the acceleration of joins. The dataflow between the three accelerators of AQUOMAN is fixed - the generality and programmability of AQUOMAN comes from the predicates the *Row Selector* applies, the functions the *Row Transformer* computes, and the operators the *SQL Swissknife* runs.

Architecturally speaking, the *Row Transformer* directly streams the intermediate table to the *SQL Swissknife* without materializing it in DRAM. In the benchmark we evaluated, this drastically reduced the need of DRAM for AQUOMAN.

V. Programming AQUOMAN

Software Interface: As shown in Figure 3, AQUOMAN is located inside the flash drive, so has direct data access to the NAND flash arrays. AQUOMAN and the x86-host can both access NAND flash simultaneously via a *flash controller switch* inside the flash device, which fairly arbitrates flash commands of *page_read, page_write, block_erase*. User-level applications can access the flash drive via legacy operating I/O stack, such as filesytem and block device drive.

In addition to legacy I/O path, AQUOMAN-enabled software can also send AQUOMAN programs to AQUOMAN inside the flash drive, which directly reads the required database files, executes the program and returns their result to the host.

In general a SQL query is compiled to a graph of *Table Task*(s). We first describe the structure of a *Table Task*:

- `table` specifies the input table of the *Table Task*.
- `maskSrc` specifies the source of the row processing masks, which is generated by a *Row Selection Program*. It can come from the *Host software*, or from AQUOMAN DRAM if produced by a previous *Table Task*.
- `rowSel` specifies a *Row Selection Program*. This selection mechanism can only compute single column predicates, but it provides a fast layer of selectivity to avoid having to stream all the data to later stages.
- `rowTransf` specifies a straightline *Row Transformation Program*, which is mapped over all the rows to transform each one into a row of the new intermediate table. The columns of the intermediate table may be different from those of the source table.
- `operator` specifies a reduction function in the *SQL Swissknife* as an SQL operation on the output table of *Row Transformation Program*. There are seven operators with self-explanatory names: `TOPK`, `SORT`, `AGGREGATE_GROUPBY`, `AGGREGATE`, `NOP`, `MERGE` and `SORT_MERGE`.
- `Output` specifies the output destination of the *Table Task*, which can be either AQUOMAN or the Host.

For simple queries such as the *Aggregate Group-By* query of Fig 1, it should be clear from the previous section that only one table task is needed.

For more complex queries, AQUOMAN programs can have multiple *Table Task*s, each of them run sequentially using an SQL operator in *SQL Swissknife* that consumes the data left by the previous *Table Task* in the AQUOMAN's DRAM (See Sec. VI-D).

```
auto tabletask_0 = TableTask{
  .table     = "inventory",
  .maskSrc   = RowSelectionProgram,
  .rowSel    = [predicate: category == "shoes"],
  .rowTransf = [in: inventoryID][out: inventoryID],
  .operator  = NOP,
  .output    = AQUOMAN_MEM_0};
auto tabletask_1 = TableTask{
  .table     = "sales_transactions",
  .maskSrc   = RowSelectionProgram,
  .rowSel    = [predicate: saledate> 2018-03-15'],
  .rowTransf = [in: inventoryID][out: inventoryID];
  .operator  = SORT_MERGE[with AQUOMAN_MEM_0],
  .output    = AQUOMAN_MEM_1};
auto tabletask_2 = TableTask{
  .table     = "lineitem",
  .maskSrc   = AQUOMAN_MEM_1,
  .rowSel    = [NOP]
  .rowTransf = [in: price][out: price];
  .operator  = AGGREGATE
  .output    = Host};
```

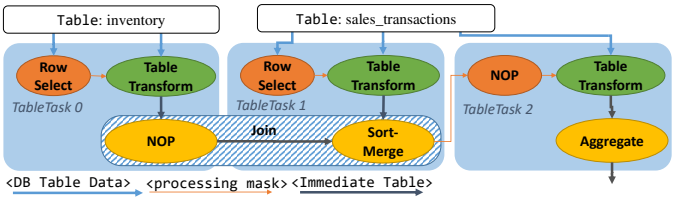

Fig. 5: JOIN query *Table Task*s and their data-flow graph

For example, to accelerate the join query (Fig. 4), the user can create the three *Table Task*s and the associated dataflow graph as shown in Fig. 5.

Since executing a single *Table Task* on AQUOMAN can saturate the flash bandwidth, executing *Table Task*s sequentially is sufficient to keep up the line rate. AQUOMAN records the intermediate results for the join in its DRAM.

VI. AQUOMAN Microarchitecture

To execute a *Table Task*, AQUOMAN first configures the *Row Selector*, the *Row Transformer* and the *SQL Swissknife* using the parameters of the first *Table Task* in the task queue. Before processing a *Row-Vector ID*, the *Row Selector* reserves a Row-Mask Vector slot in the Row-Mask Vector Array in circular order. It notifies the *Row Transformer* by sending it the *Row-Vector ID*.

The *Row Transformer* collects the *Row Vector*s of the base table, and applies a table transformation on it to produce the *Row Vector*s of the intermediate table. The *Row Transformer* then releases the slot in the Row-Mask buffer and passes the *Row Vector*s of the intermediate table to the *SQL Swissknife* to apply the specified SQL operation on the intermediate table. The output is written into AQUOMAN DRAM.

The AQUOMAN runs the three accelerators simultaneously in a pipeline fashion, as long as it can reserve a slot in the row-mask vector array. The maximum number of in-flight *Row-Vector ID*s is determined by the depth of the flash command queue, which determines the size of the Row-Mask Vector Circular Buffer. For example, for a flash controller with a command queue of depth of 128, the Row-Mask Vector Circular Buffer needs to hold a maximum of 128×8K rows of 1-byte elements or 32K 32-element *Row Vector*s.

A. Row Selector

The *Row Selector* is a *vector* unit in charge of evaluating the predicate for selection. It accepts predicates in the form: $Pr = F(CP_0, \ldots, CP_{n-1})$, where CP_i is a comparison or an equality to a constant for the value in column i, and F is a simple boolean function. For example $(price > 25)\&(data < 2019-11-26)$ is representable with $F = \&$ and $CP_0 = price > 25$ and $CP_1 < 2019-11-26$. The maximum number of permissible CP terms in a filter predicate is determined by the number of *Column Predicate Evaluator*s; 4 to 6 evaluators are enough for most of the filter predicates in TPC-H. When the *Row Selector* cannot compute a predicate, e.g. predicates which require more than one column, or regular-expression filtering, it forwards them to *Row Transformer*, the next stage in data-flow.

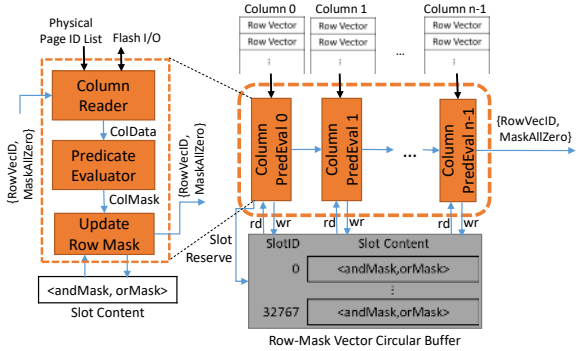

Fig. 6: Architecture of Row Vector Selection

B. Row Transformer

The *Row Transformer* has three components: the *Table Reader*, the *Row Transformation Systolic Array* and the *Mask Reader*, as shown in Figure 7.

The Table Reader initiates reading the flash drive when it receives a *Row-Vector ID* from the *Row Selector*. It skips reading a flash page if all its *Row-Vector ID*s are marked as zero in the bitvector mask. The Table Reader streams out *Row Vector*s to the *Row Transformation Systolic Array* in increasing order of *Row-Vector ID*s; within each *Row-Vector ID*, streaming is done from the leftmost column to the rightmost one.

Inside the Table Reader there is also a *Regular-Expression Accelerator*. It pre-processes variable-sized (string) columns to a one-bit column (true/false). The accelerator has a 1MB memory to store the strings of the column. 1MB is sufficient to cover many cases where the strings have a small domain, for example, the "country name" column.

The *Row Transformation Systolic Array* applies a *mapping* function to each row of the input table to produce an intermediate output table. That is, only column data of the same row are taken together to calculate columns of a new row.

The *Row Transformation Systolic Array* is a systolic architecture where the transformation function implied by a query is mapped to an array of PEs. Since the Table Reader streams out *Row Vector*s per *Row-Vector ID* in a fixed order, we can draw a data-flow graph of transformation steps from the input columns to output columns. For example, the mapping of the data-flow graph for the query in Fig. 9 is shown in Fig. 10. An AQUOMAN compiler can balance transformation data-flow graph by inserting PASS nodes (NOPs), so that it can be mapped to the PE. It can share common subexpressions used in computing several output columns by inserting FORK nodes (`Copy` Instruction). The compiler must maintain the invariant that the nodes of the compiled data-flow graph can only have data transfers to their south and/or east neighbor(s). In particular, no cycles are allowed in the dataflow graph.

Each PE performs transformation steps for multiple output columns in a circular schedule. It also produces new *Row-Mask Vector*s for filtering (sub)predicates which have not been processed by the *Row Selector*. The Mask Reader then merges the old *Row-Mask Vector* (produced by the *Mask Reader*) and the new *Row-Mask Vector* and passes it to *SQL Swissknife*. Finally, it releases the slot in the Row-Mask Vector Circular Buffer.

Each processing engine (PE) in the *Row Transformation Systolic Array* is a simple 4-stage integer arithmetic vector processor with no branch instructions or data memory (Fig. 8). It implements a simple 32-bit instruction set described in Table II. Each PE has 7 general purpose registers (`rf[1]`,..., `rf[7]`), an operand fifo (`opReg`). Finally it has a special fifo, which can be accessed as a register (`rf[0]`), hardwired to be read as input fifo and written into as the output of the PE.

Opcode	AluOp	Descr.
`Pass`		`rf[rd]<=rf[rs]`
`Copy`		`rf[rd]<=rf[rs]; opReg<=rf[rs]`
`Store`		`opReg<=rf[rs]`
`ALU(Imm)`	Add	`rf[rd]<=rf[rs] + <OpReg|imm>.`
	Sub	`rf[rd]<=rf[rs] - <OpReg|imm>`
	Mul	`rf[rd]<=rf[rs] * <OpReg|imm>`
	Div	`rf[rd]<=rf[rs] / <OpReg|imm>`
	EQ	`rf[rd]<=rf[rs] == <OpReg|imm>`
	LT	`rf[rd]<=rf[rs] < <OpReg|imm>`
	GT	`rf[rd]<=rf[rs] > <OpReg|imm>`

TABLE II: PE Instruction Set

The instruction memories of the PEs are initialized by the *Table Task*. Since there are no branches, the program counter (PC) will always increment by 1 and roll back to 0 at the end of the program. The size of the instruction memory of each PE should be bigger than the number of nodes in the transformation diagram, which equals the number of input columns to be transformed.

Once an instruction is fetched and decoded, the input *Row Vector* is read either internally from the Register File or externally (`rs==0`). The *Row Vector* is placed either in an operand register waiting for the second operand, or sent to a pipelined ALU with its other waiting operand. The Execute

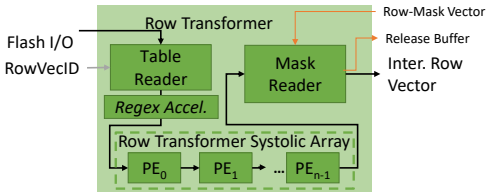

Fig. 7: Row Transformer Architecture

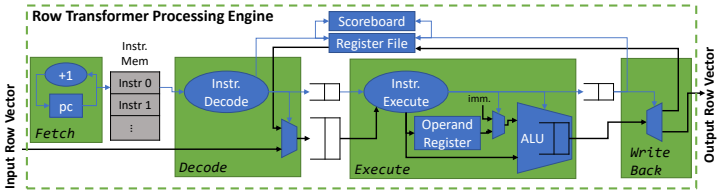

Fig. 8: Micro-architecture of a Row Transformer PE

```
SELECT l_quantity as qty,
  l_extendedprice as base_price,
  l_extendedprice*(1-l_discount) as disc_price,
  l_extendedprice*(1-l_discount)*(1+l_tax) as charge,
FROM  lineitem WHERE l_shipdate <= date '1998-09-01';
```

Fig. 9: SQL Query Example for Table Transformation

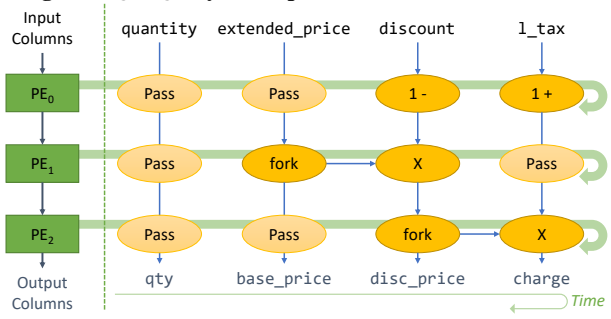

Fig. 10: Data-Flow Execution Diagram of Table Transform

stage performs the operation and in the write-back stage, the output of the ALU is either written in the register file or streamed out (`rd==0`). The Register File is used only for data passing vertically between nodes when multiple nodes of a vertical slice of the graph are mapped to a single PE. Such a case happens only when the number of PEs exceeds the number of horizontal layers of the data-flow graph.

C. SQL Swissknife

The *SQL Swissknife* is configured by the *Table Task*, which takes the intermediate table output from the *Row Transformer*, and applies one of the SQL (sub)operations listed in Section V. Inside the SQL Swissknife is an array of accelerators, whose connection to the external input is configured by the *Table Task* (Fig. 11). When *Row Vector*s are streamed in, they are tagged with a Column ID which is needed for processing a table of more than one columns (e.g the input table of an Aggregate GroupBy). Each SQL operation is mapped to its corresponding accelerator(s). SQL sub-operators of `SORT`, `MERGE` and `SORT_MERGE` are mapped to two serially linked accelerators: the Streaming Sorter and the Merger. For `SORT` and `MERGE`, one of them is configured as a NOP.

New SQL operation accelerator can be added into SQL Swissknife with or without DRAM access as needed. In our current version of the *SQL Swissknife*, only the Streaming Sorter and Merger are connected to the DRAM.

Aggregate GroupBy*:* The Aggregate GroupBy accelerator handles grouping rows of the same group identifier into summaries of aggregation attributes of `sum`, `min`, `max`, and `cnt`. It does local *Aggregate Group-By* operation per *Row-Vector ID*, and then scatters and updates the local group aggregates to the corresponding global aggregates stored in SRAM.

As shown in Figure 12, the Aggregate GroupBy accelerator separates *Row Vector*s of columns into two different streams using their Column IDs. If *Row Vector*s are used for identifying groups, they are sent to the *Column Zipper*, otherwise they are sent to the *Reduce-By-Group-Number* block, waiting for their groupIDs to be assigned.

Column Zipper zips multiple *Row Vector*s of the same *Row-Vector ID* into a super *Row Vector* named the *Group Identifier Vector*. The *Group Number Assign* component assigns it a Group Number using a hash-table of 1024 buckets. Each bucket can hold at most one group identifier of maximum size of 16B. New group numbers are assigned in an increasing order from 0 to 1023. In case of a hash collision of two group identifiers, one group is kept and the other one is marked as a spill-over group, which is sent to x86 host for processing. (more on this in Section 6.5)

After a Group Identifier Vector is given a Group Number, it is sent to the *Reduce By Group Number* block, in which its corresponding *Row Vector*(s) are reduced per group. The reduction results of `sum`, `min`, `max`, `cnt` are scattered into an SRAM and accumulated with the global aggregates indexed by group number. Each aggregate slot can store aggregates for 8 individual columns.

Since the SRAMs are expected to scatter/gather a maximum of 32 addresses per request, we have partitioned the SRAM into 32 partitions by striping the address space, allowing bigger bandwidth through banking. If addresses per scatter or gather request are uniform, we can pipeline the requests without many memory stalls.

TopK*:* The TopK accelerator takes in a stream of *Row Vector* from a table and keeps the biggest k rows of the stream. In software, the *TopK* operation is computed using minHeap, which cannot be easily pipelined in hardware. Instead, we use a chain of *Vector Compare-And-Swap* blocks (VCAS) to store the k biggest elements, as illustrated in Figure 13. Each VCAS stores n elements where n is the input vector size. When a vector of size n is fed into a VCAS, VCAS compare-and-swaps it with the n elements stored inside the VCAS, where the bigger half of $2n$ elements are kept, and the smaller half is streamed out. We can daisy-chain k/n VCAS to keep the top k elements.

Before sending it into the chain of VCAS, the input vector is first sorted using a pipelined bitonic sorter. This is done because the pipelining of VCAS operation for sorted vectors can be done more efficiently, as shown in Figure 13. Each VCAS operation of two sorted vectors of size n can be divided into n steps of compare-and-swap element-wise, as shown in

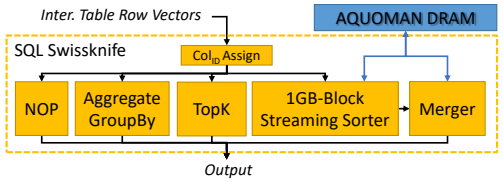

Fig. 11: SQL Swissknife Architecture

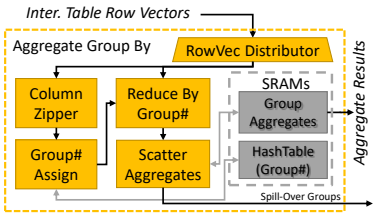

Fig. 12: Aggregate-GroupBy Accel.

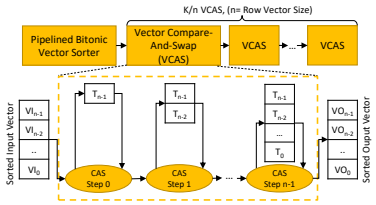

Fig. 13: TopK Accelerator

Algorithm 1. The *i*th element-wise CAS step generates a partial

Algorithm 1 Vector Compare-and-Swap

Variables: *InVec*: Input Vector sorted in ascending order
TopVec: Top-n Vector sorted in ascending order

$tailIn = tailTop = n-1$
for i in $0..n-1$ **do**
  if $InVec[tailIn] > TopVec[tailTop]$ **then**
    $swap(InVec[tailIn], TopVec[tailIn])$
    $tailIn = tailIn - 1$
  else
    $tailTop = tailTop - 1$
  end if
end for

result of the top-i vector which can be consumed by the ith step of the next input vector immediately. Therefore, a VCAS hardware can be pipelined properly. Each ith pipeline stage of the VCAS takes up reasonably small hardware resources with one pair of `i-to-1` muxes for compare, and one pair of `i-to-1` muxes for data update.

Merger: The Merger accelerator outputs the intersection of two sorted list. The Merger accelerator first merges two sorted list into one sorted list using *2-to-1 Merger*, and then passes it through an *Intersection Engine* where the non-intersected part is dropped (Figure 14).

In case of duplicate values in the input sources, the merger always tries to alternate the input sources. This way the Intersection Engine only needs a look-ahead of one to decide if it should drop a value or not. Indeed if in the final sorted stream two consecutive values are equal but not coming from the same source, one of them can be dropped knowing that the same value from the other source could not arrive later.

Inside the 2-to-1 Merger, we have the *Vector Compare-And-Swap Engine* which does the merging, and a *Scheduler* which decides which input vectors of the two sorted streams should be fetched. Since items of each input vector are sorted and the input vectors per data stream are sorted, the Scheduler only needs to compare the top items of the two input vectors and send the input vector with the smaller top item to VCAS.

1GB-Block Streaming Sorter: The 1GB-Block Streaming Sorter takes an unsorted stream of input vectors, and outputs a stream of sorted 1GB blocks. The Streaming Sorter consists of a *Pipelined Bitonic Sorter* which sorts 64-byte input vectors, and merge 2^{24} 64 bytes vectors into a 1GB sorted stream using three layers of 256-to-1 Mergers (Figure 15).

The first two layers of the 256-to-1 mergers merge 256 64B-blocks to a 16KB block, and 256 16KB-blocks to 4MB-block respectively. They store the immediate results on SRAMs. The last layer merges 256 4MB-blocks to 1GB block using DRAM. The SRAMs and DRAM need to be duplicated per layer to maintain the line-rate of input stream. If the sorter had enough DRAM, it can sort 256GB by folding the last 256-to-1 merging step at the half of the streaming speed.

Each 256-to-1 Merger is constructed using a binary tree of 2-to-1 mergers which were introduced in Section VI-C. Since the average of utilization of 2^i 2-to-1 mergers at the same depth i of the binary tree is only 1, we make 2-to-1 mergers at the same depth share the same VCAS component capable of keeping multiple contexts. In this way, we can decrease the size of N-to-1 merger from $O(2N-1)$ to $(O(logN))$ while still able to keep up with the input rate.

To perform sort-merge join, both join-key columns don't have to be totally sorted. As long as one column is totally sorted, a partially-sorted second column can be merged with it at the cost of re-streaming the first one for every 1GB of data stream. This can cause more than 1GB DRAM reads per 1GB flash reads, but is OK because DRAM is an order-of-magnitude faster than flash. In many cases, AQUOMAN doesn't even need to sort the first column, since primary keys are already stored by MonetDB in its internal representation.

D. AQUOMAN *Memory Management*

Because of the fixed dataflow pipeline of AQUOMAN, a SQL query often needs to be broken into multiple *Table Task*s which are executed on AQUOMAN sequentially. AQUOMAN stores the intermediate tables produced by each *Table Task* on DRAM and merge them using subsequent *Table Task*s. AQUOMAN's memory management system only keeps the row indices of tables and join keys in DRAM to compute *multi-way joins*, which allows us to keep the DRAM footprint small. AQUOMAN memory management does not buffer the results of *Aggregate Group-By* and *TopK* operators, because such operators are typically at the end of an SQL execution plan. When such operators are not the last operator of the query, we cannot off-load the part of the query following the *Aggregate Group-By* or *TopK* operator. Such cases are uncommon and AQUOMAN can often accelerate even partially offloaded queries (see Sec. VIII-B).

A RowID column provides index to rows of a table. Such a column is implicit and does not need to be stored in DRAM or flash. A multi-way join is decomposed into two-way joins, where each two-way join is executed using a *sort-merge join* expressed by two *Table Task*s. Each data flow arc which goes into *sort* and *sort-merge* operations carries key-value pairs, where the key field is used for sorting and merging, and the value field has the RowID representing where the join key is

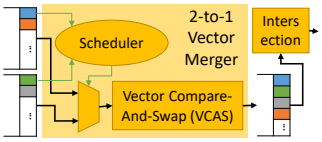

Fig. 14: Merger Architecture

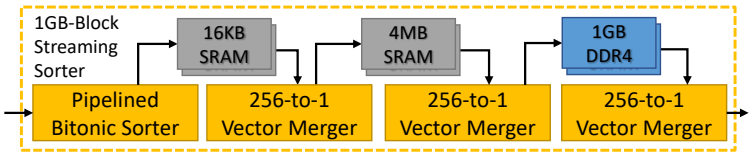

Fig. 15: Streaming Sorter Architecture

read from. The *Table Task*s of each two-way join produces two intermediate tables. The intermediate tables produced by sort *Table Task*s are consumed by their subsequent sort-merge *Table Task*s, and can be garbage collected immediately. The intermediate tables produced by sort-merge *Table Task*s store backward pointers, *i.e.* RowIDs, which are needed for constructing the final result of a multi-way join. And they are stored for the entire lifetime of a multi-way join query.

AQUOMAN also deploys MonetDB-specific optimizations to save memory. MonetDB uses RowIDs to represent the primary keys of tables internally, and for each foreign key column it materializes an additional column of RowIDs referring to the primary keys. MonetDB uses RowIDs to perform join whenever is possible. AQUOMAN is aware of the internal structure used by MonetDB and avoids loading the RowIDs to DRAM whenever possible. Such an optimization opportunity arises when all the primary keys of a table are used for a join operation, i.e. no row of the table has been deleted or filtered out. No join operation is required by AQUOMAN in this case since all foreign keys of the second table are guaranteed to find their matching primary keys. Therefore we can avoid using DRAM and directly construct the join result using the materialized RowIDs on flash.

E. Suspending Query Processing on AQUOMAN

There are several reasons why a query may not be completely processed by AQUOMAN:

1. A query has an *Aggregate Group-By* operator in the middle of an execution plan, which breaks references to the base tables on flash.
2. A query does regular-expression filtering on a variable-sized string column which requires pointer references to a string heap file. When there are many unique strings, such string operations cause random reads to the string heap on the flash and is unsuitable for processing by AQUOMAN.
3. An *Aggregate Group-By* operator in a query generates more groups than what AQUOMAN's SRAM can accommodate.
4. A multi-way join operation in a query produces intermediate tables that exceed AQUOMAN's DRAM capacity.

Conditions 1 and 2 can be detected by examining the query plan, and AQUOMAN can simply suspend processing the query at the appropriate point and pass the intermediate table of results to the host, which can resume processing the query. Since the host will need to access the AQUOMAN SSD when it resumes the query processing, that SSD remains essentially unavailable to AQUOMAN until the query to AQUOMAN has been processed completely. Conditions 3 and 4 can be detected only during query execution. If the database system has an estimate for the size of the intermediate data structure for a specific dataset, it may decide not to offload a part of the query to AQUOMAN. Otherwise, AQUOMAN may use the suspension strategy described next.

For large *Aggregate Group-By* operator, AQUOMAN computes all the hashes but performs the accumulate operation on some buckets in AQUOMAN, while the accumulation for the "spillover" buckets is performed by the host. To not slow down AQUOMAN, the host needs to keep up with the spills generated by AQUOMAN.

For multi-way *Join*s, when the AQUOMAN DRAM becomes full, it keeps sorting 1GB-data-blocks and sends them to the host via DMA. The host completes the join operation by merging these sorted blocks with the sorted data stored in its DRAM.

A natural question to wonder about is how common are these suspensions. As we will show in Sec. VIII, 14 out of the 22 queries of TPC-H can be offloaded completely to AQUOMAN with sufficient DRAM. Queries (11,17,18,22) encounter *Aggregate Group-By* operator in the middle and thus, had to be suspended; all except Q22 benefited by partial offloading. There was no benefit to offload queries (9,13,16,20) because they involved regular-expression filtering on a string column.

Seven queries caused spillovers in the *Aggregate Group-By* operation. Only Q18 caused a significant spillover (required ~1.5 billion buckets while AQUOMAN has only 1024 buckets!). Partial offloading of Q18 was still profitable, assuming the host could perform ~200 millions memory lookup-and-accumulates per seconds. With 40GB DRAM in AQUOMAN there were no suspensions due to multi-way *Join*s. A conservative approximation of the effect on performance of memory limitations is discussed in the Evaluation Section VIII.

VII. AQUOMAN Implementation

AQUOMAN was first implemented on an FPGA, although soon afterwards we discovered several reasons which prevented us from evaluating most TPC-H queries on the FPGA prototype:

- Our FPGA evaluation board has only 4GB of DRAM, which is not big enough to evaluate multi-way joins that generate bigger intermediate tables.
- AQUOMAN with the *Sorter* exceeded the total area of the FPGA in BlueDBM. Our bigger FPGA, VCU118, is not compatible with the FMC port of the custom flash card in BlueDBM.
- A robust regular-expression accelerator, which has been done previously [21], is also needed for string columns but required significantly more implementation effort than this project justified.
- A full-blow compiler is needed to generate *Table Task*s for FPGA evaluation; manual compilation effort is too high for

most TPC-H queries. (Investment in such a compiler would be justified only after the efficacy of AQUOMAN has been established.)

In order to evaluate more queries and to properly evaluate the AQUOMAN architecture, we also developed a trace-base AQUOMAN simulator and fully integrated it in the MonetDB's software stack . We validated some of our simulation results on the FPGA proptotype (Sec. VIII-D)

FPGA Prototype: We implemented AQUOMAN on BlueDBM [30], where a hardware-accelerated storage device is plugged into the PCIe bus of 12-core Xeon X5670 machine. Each storage device consists of a Xilinx Virtex Ultrascale FPGA development board, VCU108, attached to 1TB of open-channel NAND flash array capable of 2.4GB/s read access and 800MB/s write access. The Xilinx VCU108 FPGA also provides 4GB of DDR4 memory for a maximum bandwidth of 36GB/s.

We synthesized the *Sorter* and the rest of AQUOMAN on two different FPGAs because together their area exceeded the capacity of the VCU108 FPGA. In our AQUOMAN implementation (Table III) the *Row Selector* has 4 Column Predicate Evaluators, and the *Row Transformer* has 4 processing engines each with 8 instructions. Our design meets the timing requirement for 125MHz and provides 4GB/s processing rate for AQUOMAN.

Module Name	LUTs	Flip-Flops	RAMB36	DSP48
Row Selector	42023	36725	0	0
Row Transformer	47859	29660	0	256
SQL Swissknife (w/o sorter)	95077	76823	140	0
FlashPageBuffer	14087	17143	228	0
RowMask	190	41	58	0
VCU108 Total	302398 (56%)	273245 (24%)	448 (26%)	256 (33%)

TABLE III: AQUOMAN resource usage on VCU108

1GB-Block Hardware Sorter: We synthesized the 1GB-block streaming sorter for four data types: 32/64-bit integers, and key-value pairs of 32/64-bit integers. All designs were synthesized with a 512-bit data path and met the timing requirement for 200MHz on Xilinx UltrascalePlus VCU118. Flip-Flops usage was around 40% for each configuration (see Table IV).

Element Type	LUTs	RAMB36	URAM
`uint32`	855867 (72%)	1133 (52%)	256 (27%)
256-to-1 Merger to 16KB	240567 (20%)	177 (8%)	0 (0%)
256-to-1 Merger to 4MB	263610 (22%)	291 (13%)	256 (27%)
256-to-1 Merger to 1GB	261400 (22%)	505 (23%)	0 (0%)
`uint64`	925572 (78%)	1133 (52%)	256 (27%)
`kv<uint32,uint32>`	720183(60%)	1133 (52%)	256 (27%)
`kv<uint64,uint64>`	900087(76%)	855 (40%)	256 (27%)

TABLE IV: Streaming Sorter resource usage on VCU118

The area of *Sorter* and AQUOMAN together exceeds the Xilinx VCU118 capacity by 2% but we are confident that with a few area optimization we can fit both of them on a VCU118. Unfortunately VCU118 is incompatible with the custom flash card in BlueDBM.

Although AQUOMAN uses 64-bit key and value pairs as the *Sorter* configuration, we evaluated the streaming sorter for all sorter configurations. As expected, all configurations have the same throughput. Table V summarizes the performance of the *Sorter* for different input lengths and sortedness using a traffic generator. Hence our *Sorter* meets the goal of keeping up with AQUOMAN processing bandwidth (4GB/s).

Input Length (GB)	Input Sortedness: Sorted	Reverse Sorted	Random
1	4.4 GB/s	4.4 GB/s	6.2 GB/s
10	7.9 GB/s	7.9 GB/s	11.0 GB/s
100	8.5 GB/s	8.5 GB/s	11.9 GB/s
1000	8.6 GB/s	8.6 GB/s	12.0 GB/s

TABLE V: 1GB-Block Streaming Sorter Throughput

AQUOMAN *Simulator:* We implemented a trace-based AQUOMAN simulator which is fully integrated in MonetDB 11.27.9. In MonetDB the software translates the SQL execution plan into a customized middle-layer language, Monet Assembly Language(MAL), which is then later optimized and interpreted [25]. We implemented the AQUOMAN simulator by extending MAL to allow instrumenting traces for AQUOMAN *Table Task*s. The AQUOMAN simulator does not execute *Table Task*s, but executes the original SQL plan expressed in MAL and collects AQUOMAN traces such as flash traffic, AQUOMAN memory footprint, and sorter usage. The AQUOMAN simulator assumes a flash drive of 8KB page access granularity and 2.4GB/s flash read bandwidth, one streaming sorter and one regular expression accelerator with 1MB cache for string heap. The specifications of flash drive and streaming sorter in AQUOMAN simulator are the same with the ones in the AQUOMAN FPGA prototype in Section VII. We assume as big Row Selector and Row Transformer as needed as their small relative sizes compared to the sorter as shown in Section VII. When a multi-way join query exceeds AQUOMAN memory size, we assume that the host processes "handed-off" sub-query at the same speed as the baseline solution, which is a conservative assumption.

In the SQL frontend, we modified MonetDB's query planner to identify *Table Task*s in the query execution plan tree, and mark relevant nodes as AQUOMAN nodes which are targets for offloading. We also changed MonetDB's SQL plan to MAL compiler, such that AQUOMAN tracing instrumentation will be automatically inserted on identified *Table Task*s. The total execution time of a query with AQUOMAN simulator is calculated by the sum of AQUOMAN execution time based on the traces and non-AQUOMAN nodes' execution time processed by MonetDB.

VIII. Evaluation

A. Experiment Setup

Evaluation Data-set: We used the TPC-H synthetic data-set with a scaling factor of 1000, generating **1TB** of tables. We loaded the data-set on MonetDB-11.27.9, whose column files are the inputs for AQUOMAN.

Baseline Setup: We ran MonetDB (11.27.9) with two setups S and L to represent two different machine sizes (Table VI). The baseline used five 1TB Samsung 970 EVO m.2 SSDs capped at 2.4GB/s, to match the bandwidth of the BlueDBM storage device. Such a setup was needed for a fair baseline 1) to mitigate side-effects of garbage collections with over-provisioned capacity and 2) to provide 2.4GB/s average access

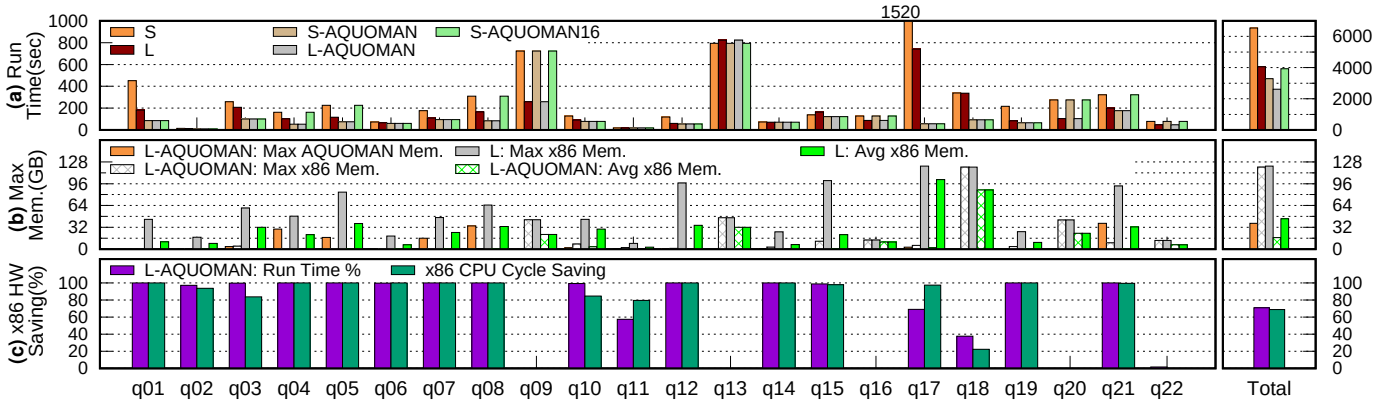

Fig. 16: TPC-H SF-1000 AQUOMAN Performance Profiling

bandwidth unavailable in a single off-the-shelf SSD. When MonetDB run out of DRAM for intermediate tables, it can still process queries effectively by using its own disk-swap management, which exploits fast sequential SSD writes.

MonetDB does not implement a page buffer pool for caching hot pages, instead it relies on Linux's LRU-based page cache to take advantage of page locality. For 1TB dataset, we observed that Linux's page cache on a 128GB DRAM is ineffective for TPC-H queries. In fact, for MonetDB hot runs are slightly slower than cold runs, even though both runs experience a similar level of page-cache misses. We hypothesize that the cost of finding misses in a fully populated page-cache is larger because the operating system needs to traverse a bigger page-cache index structure to find nothing. Therefore our evaluations assume cold page cache.

x86 Setup	HW Threads	x86 DRAM
S (Small)	4 Threads	16GB
L (Large)	32 Threads	128GB

AQUOMAN Setup	DRAM
AQUOMAN	40GB
AQUOMAN16	16GB

TABLE VI: x86 Host and AQUOMAN Disk Setup

AQUOMAN *Setup:* All TPC-H queries are evaluated on the AQUOMAN simulator, which has two setups: AQUOMAN with 40GB memory and AQUOMAN16 with 16GB memory (Table VI). The AQUOMAN implementation on FPGA could be used for evaluating only a few TPC-H queries because of the reasons discussed earlier (Sec. VII). However, the FPGA implementation was very useful to validate the AQUOMAN simulator (Sec. VIII-D).

B. AQUOMAN *TPC-H Evaluation*

We compare the performance of MonetDB running on a system with ordinary SSDs with a system where the ordinary SSDs are replaced by AQUOMAN SSDs. For an extensive coverage of the design space, two host machines S, L (Fig. VI) are used, each with and without AQUOMAN disks. We also paired the small host system S with AQUOMAN16 disk, which has 16GB of in-storage DRAM.

We first examine the runtime of each query, including the breakdown of the time spent on AQUOMAN and the host. We then perform a similar analysis for the memory footprint.

Run Time: The queries run time for the different systems are presented in Fig. 16(a), while the fraction of processing time each query spent on AQUOMAN for system (L) is shown in Figure 16(c) . On average, 71% of the CPU time can be saved by AQUOMAN when it is added to system L. Note that AQUOMAN actually speeds up many queries, on average a 1.5X-2X speed-up over the baseline in Fig. 16(a). Still there are some outliers, queries (17,18) show up to 13X speed-ups for system L, while others show none. It is important to realize that AQUOMAN cannot speed up a query if it is IO bound in the baseline system. In such cases it can only save host resources. For example, we found that two queries (6,14) can be almost completely off-loaded to AQUOMAN but show little speedup because they are disk-bound on the baseline systems.

For 14 out of the 22 queries are off-loaded to AQUOMAN nearly 100% of the time. Even when AQUOMAN can only do a part of the query, its resulting benefits can be significant. For example, the runtimes of Q17 and Q18 decrease significantly because the part that is off-loaded happens to execute sequentially on the host, effectively using only one hardware thread.

The reasons for queries to be suspended early were discussed in Sec. VI-E. As we said earlier queries (17,18,22,11) corresponds to cases with an early *Aggregate Group-By* node in the execution plan. All of them except Q22 do enough processing on AQUOMAN to show speedups. Queries (9,13,16,20) represent the cases where the size of the string heap does not fit in AQUOMAN and so have to be completely handled by the host.

Overall when a host machine replaces its SSDs with an AQUOMAN disk, it can save on an average 71% of the CPU time for TPC-H queries running on system L.

Memory Footprint: Figure 16 (b) shows the maximum and average memory resident set size (RSS) of AQUOMAN and the system-L baseline, respectively. When a query is adequately offloaded, AQUOMAN reduces host memory footprint significantly except when it has to aggregate on a huge number of groups in the host as for Q18 (See Sec. VI-E). AQUOMAN has 20~128GB smaller memory footprint than the baseline even when most of the query is processed by AQUOMAN. The memory saving is primarily because of the streaming model of *Table Task*s and only keeping row IDs in the DRAM. The maximum memory requirement for the TPC-H benchmark by AQUOMAN on 1TB data-set is 40GB. In fact when equipped

with 16GB DRAM, only 4 queries (4,5,8,21) are affected and AQUOMAN can offload 12 of 22 TPC-H queries profitably. We also note that while AQUOMAN reduces the average DRAM used significantly (by a factor 3), the maximum DRAM needed is left almost unchanged. Indeed an important part of Q18 has to be processed by the host, and it requires almost 128GB of DRAM.

C. Advantages of AQUOMAN

In the previous section we have shown significant hardware threads and memory savings. In this section we propose a way to visualize the benefits from those savings (70% of CPU and 60% of the average memory). As a first approximation, Fig. 16(a) shows that running the entire TPC-H benchmark on a 32-cores machine with 128GB DRAM (system L) is on average 1.6X as fast as running the same benchmark of on a 4-core machine with 16GB DRAM (system S). However, replacing the SSD of the small machine by an AQUOMAN augmented SSD (S-AQUOMAN16) bridges that gap completely!

This comparison does not evaluate the opportunity for the system to run many queries in parallel, which may show different results because of inter-query data locality and parallelization. Evaluating such a parallel system requires a very different setup than what we have presented in this paper.

D. Validating simulation results on FPGA

The AQUOMAN FPGA prototype has the key AQUOMAN components: *Row Selector*, *Row Transformer*, and *SQL Swissknife* with a high-performance *Sorter* but is limited by 4 GB of DRAM. We evaluated a subset of TPC-H queries using the AQUOMAN FPGA prototype to validate some of our simulation results.

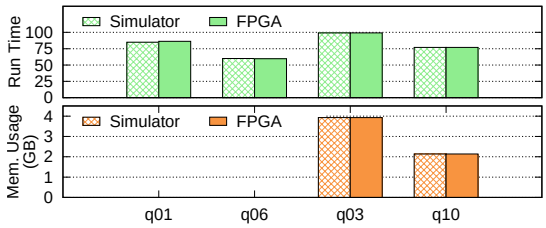

Fig. 17: TPC-H queries on FPGA prototype

We picked two classes of TPC-H queries and hand-coded them to *Table Task*s to execute on the FPGA prototype. The first type of queries (1,6) have no join operations. Those queries are evaluated end-to-end and produce the same query results as the MonetDB software. The second type of queries (3,10) are multi-way join queries but need less than 4GB AQUOMAN DRAM. Since currently we cannot fit AQUOMAN and the *Sorter* on a single FPGA, we are unable to run multi-join queries end-to-end on AQUOMAN. In the AQUOMAN design, *Table Task*s are executed sequentially, so we executed each *Table Task* of a query individually and summed up the execution times for the end-to-end query run time. For the *Table Task*s that involve *Sort*, we use a traffic generator instead of real data, throttled at the same speed as AQUOMAN's flash card (2.4GB/s). This is because our flash card is incompatible with Xilinx VCU118's newer version of the FMC+ connector. None of the evaluated queries had regular expression selections.

For each query, we compared the run time and memory usage of the FPGA prototype with those of AQUOMAN simulator (Fig. 17). We can see the FPGA prototype has similar run times and the same memory usage as the AQUOMAN simulator, which validates the basic performance modeling of AQUOMAN.

Compared to FCAccel [50], our FPGA evaluation used 10X the dataset size and evaluated more queries including fully-offloaded joins. Therefore we cannot provide direct query run-time comparison, but we can compare in terms of rows/sec with FCAccel's evaluation. AQUOMAN's FPGA performance is competitive to that of FCAccel. For the straightforward filter-and-aggregate with high selectivity (Q6), AQUOMAN has similar throughput (100.5M rows/s vs. 111M rows/s). When a query has low selectivity and requires more computation, such as row transform and *Aggregate Group-By* in Q1, AQUOMAN is 2.5X better than FCAccel (69M rows/s vs. 27M rows/s). This is thanks to AQUOMAN 's systolic-array design for highly-pipelined row transformation (Sec. VI-B), while FCAccel uses on multi-cycle logic designs.

IX. Conclusion

We have presented AQUOMAN, an end-to-end DBMS system solution for in-storage analytical SQL query acceleration. AQUOMAN aggressively pushes the idea of "in-storage computing" by offloading most of the query processing, including multi-way *Join*s, for terabyte data-sets. AQUOMAN is based on a novel stream-oriented microarchitecture to execute static dataflow graphs of SQL operators organized as *Table Task*s. We have built a prototype of AQUOMAN using a Xilinx VCU108 FPGA development board, and shown that it computes *Table Task*s at a 4GB/s, saturating the flash-drive bandwidth. (For power, cost and area reasons, a commercially viable version of AQUOMAN will have to be implemented using ASICs). One way to think of the savings provided by offloading queries to AQUOMAN is to imagine running SQL queries on a one-terabyte TPC-H benchmark data-set on two systems: MonetDB running on a 4-core, 16GB-DRAM machine with AQUOMAN-augmented SSDs and MonetDB running on a 32-core, 128GB-DRAM machine with standard SSDs. We have shown that, if we run queries sequentially and assume no reuse of page-cache by different queries, the two system provide the same performance. The future work on AQUOMAN requires (1) an experimental setup to evaluate parallel execution of queries and (2) distributed execution of queries whose data is spread over multiple AQUOMAN SSDs.

Acknowledgment

We want to thank all anonymous reviewers for their comments. This work is funded by Samsung Semiconductor (GRO grants) and NSF (CCF-1725303). The DGIST team is supported by the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031).

REFERENCES

[1] "Big Data Analytics On-Premises, in the Cloud, or on Hadoop — Vertica," https://vertica.com/, accessed: 2020-04-07.

[2] "Presto on Amazon EMR - Amazon Web Services (AWS)," https://aws.amazon.com/emr/details/presto/, accessed: 2020-04-07.

[3] "S3 Select and Glacier Select – Retrieving Subsets of Objects," https://aws.amazon.com/blogs/aws/s3-glacier-select/, accessed: 2020-04-07.

[4] "The column-store pioneer — MonetDB," https://monetdb.org/home, accessed: 2020-04-07.

[5] "Uber's big data platform: 100+ petabytes with minute latency," https://eng.uber.com/uber-big-data-platform/, accessed: 2020-04-07.

[6] "Amazon Redshift - Data Warehouse Solution - AWS," https://aws.amazon.com/redshift, May 2018.

[7] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar, "A many-core architecture for in-memory data processing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123985

[8] C. Balkesen, N. Kunal, G. Giannikis, P. Fender, S. Sundara, F. Schmidt, J. Wen, S. Agrawal, A. Raghavan, V. Varadarajan, A. Viswanathan, B. Chandrasekaran, S. Idicula, N. Agarwal, and E. Sedlar, "Rapid: In-memory analytical query processing engine with extreme performance per watt," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 1407–1419. [Online]. Available: http://doi.acm.org/10.1145/3183713.3190655

[9] R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wehr, "A back-end computer for data base management," *Commun. ACM*, vol. 17, no. 10, pp. 575–582, Oct. 1974. [Online]. Available: http://doi.acm.org/10.1145/355620.361172

[10] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554787

[11] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 7:1–7:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195647

[12] R. Chen and V. K. Prasanna, "Accelerating equi-join on a cpu-fpga heterogeneous platform," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 212–219.

[13] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent ssds," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465003

[14] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485945

[15] A. De, M. Gokhale, R. Gupta, and S. Swanson, "Minerva: Accelerating data analysis in next-generation ssds," in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 9–16. [Online]. Available: https://doi.org/10.1109/FCCM.2013.46

[16] D. J. DeWitt, "Direct - a multiprocessor organization for supporting relational data base management systems," in *Proceedings of the 5th Annual Symposium on Computer Architecture*, ser. ISCA '78. New York, NY, USA: ACM, 1978, pp. 182–189. [Online]. Available: http://doi.acm.org/10.1145/800094.803046

[17] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "Gamma - a high performance dataflow database machine," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 228–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671463

[18] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1221–1230. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465295

[19] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 639–651, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080233

[20] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, "phoenix: Memory speed hpc i/o with nvm," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*.

[21] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[22] R. Greenwald, M. Bhuller, R. Stackowiak, and M. Alam, *Achieving extreme performance with Oracle Exadata*. McGraw-Hill, 2011.

[23] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 153–165. [Online]. Available: https://doi.org/10.1109/ISCA.2016.23

[24] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer, "Accelerating join operation for relational databases with fpgas," in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 17–20. [Online]. Available: https://doi.org/10.1109/FCCM.2013.17

[25] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 40–45, 2012.

[26] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1202–1213, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137632

[27] Z. Istvan, L. Woods, and G. Alonso, "Histograms as a side effect of data movement for big data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1567–1578. [Online]. Available: http://doi.acm.org/10.1145/2588555.2612174

[28] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong, "Yoursql: A high-performance database system leveraging in-storage computing," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 924–935, Aug. 2016. [Online]. Available: https://doi.org/10.14778/2994509.2994512

[29] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[30] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750412

[31] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "Grafboost: Using accelerated flash storage for external graph analytics," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*,

ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 411–424. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00042

[32] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540748

[33] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 219–231. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124553

[34] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, "Extrav: Boosting graph processing near storage with a coherent accelerator," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1706–1717, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137776

[35] A. Lottarini, J. a. P. Cerqueira, T. J. Repetti, S. A. Edwards, K. A. Ross, M. Seok, and M. A. Kim, "Master of none acceleration: A comparison of accelerator architectures for analytical query processing," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 762–773. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322220

[36] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "Graphssd: Graph semantics aware ssd," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 116–128. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322275

[37] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "an analysis of persistent memory use with whisper."

[38] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 471–484. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541959

[39] J. Ouyang, W. Qi, W. Yong, Y. Tu, J. Wang, and B. Jia, "Sda: Software-defined accelerator for general-purpose distributed big data analysis system," in *Hot Chips: A Symposium on High Performance chips, Hotchips*, 2016.

[40] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid cpu-fpga databases," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 211–218.

[41] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678

[42] Z. Ruan, T. He, and J. Cong, "Insider: Designing in-storage computing system for emerging high-performance drive," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. Berkeley, CA, USA: USENIX Association, 2019, pp. 379–394. [Online]. Available: http://dl.acm.org/citation.cfm?id=3358807.3358840

[43] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 67–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685055

[44] R. Shu, P. Cheng, G. Chen, Z. Guo, L. Qu, Y. Xiong, D. Chiou, and T. Moscibroda, "Direct universal access: Making data center resources available to fpga," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. Berkeley, CA, USA: USENIX Association, 2019, pp. 127–140. [Online]. Available: http://dl.acm.org/citation.cfm?id=3323234.3323246

[45] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid cpu-fpga architectures," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 403–415. [Online]. Available: http://doi.acm.org/10.1145/3035918.3035954

[46] M. Singh and B. Leonhardi, "Introduction to the ibm netezza warehouse appliance," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '11. Riverton, NJ, USA: IBM Corp., 2011, pp. 385–386. [Online]. Available: http://dl.acm.org/citation.cfm?id=2093889.2093965

[47] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, ser. FAST'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 119–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=2591272.2591286

[48] M. Ubell, *The Intelligent Database Machine (IDM)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 237–247. [Online]. Available: https://doi.org/10.1007/978-3-642-82375-6_14

[49] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, "Relational query processing on opencl-based fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–10.

[50] S. Watanabe, K. Fujimoto, Y. Saeki, Y. Fujikawa, and H. Yoshino, "Column-oriented database acceleration using fpgas," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, April 2019, pp. 686–697.

[51] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 963–974, Jul. 2014. [Online]. Available: http://dx.doi.org/10.14778/2732967.2732972

[52] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "The q100 database processing unit," *IEEE Micro*, vol. 35, no. 3, pp. 34–46, 2015.

[53] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[54] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, ser. DaMoN '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/2771937.2771945

[55] D. Ziener, F. Bauer, A. Becher, C. Dennl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J.-S. Vogt, and H. Weber, "Fpga-based dynamically reconfigurable sql query processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 4, pp. 25:1–25:24, Aug. 2016. [Online]. Available: http://doi.acm.org/10.1145/2845087