From SQL to Database Processors: A Retargetable Query Planner

Arda Yurdakul
Computer Engineering
Bogazici University
Istanbul, Turkey
Email: yurdakul@boun.edu.tr

***Abstract*—In the literature, there are various types of hardware accelerators for managing in-memory databases. Deriving an efficient query planner for a hardware accelerator is not easy, because the architecture of each accelerator is different. SQL is a domain-specific language for describing database queries. In traditional query planning, SQL descriptions are converted to a high-level language like C, then the compiler generates the machine code. In this paper, we propose a method for obtaining query plans directly from SQL descriptions for programmable in-memory database accelerators. Given an SQL query, our method firstly converts it to a synchronous data-flow graph (SDF). Then, a set of scheduling algorithms are applied on the SDF by considering architectural specifications of the target accelerator. We also present automatically generated query plans.**

I. Introduction

With the rise of big data analytics, fast and energy efficient processing of data residing in databases have become more important than ever. It has been proved that column-store databases are more efficient than traditional row-store architecture when huge amount of data has to be queried in an ad-hoc manner, as it is the case in data warehouses [1]. Reducing the feature size of electronic systems have enabled entire database tables to exist in the main memory [2]. Computing platforms have equally benefitted from this feature reduction as we now see System-on-Chips that couple a few cores with dynamically reconfigurable Field Programmable Gate Arrays (FPGAs). Besides, software tools have also improved to make design on these platforms much easier than ever. As a result, there are hardware accelerators for column-store database management systems [2][3][4][5][7][10].

While hardware accelerators speed up the query processing, planning the queries for an accelerator has turned out to be another design problem, because each accelerator has its own architecture. Considering the way in which the query plan is utilized, we see two types of database architectures in the literature: In Query-specific Database Processors (Q-DPU), an intermediate representation is generated from the query plan. Then, the hardware accelerator is designed or generated according to this query plan [6][7][8][9]. In Programmable Database Processors (P-DPU), an accelerator is designed by using operation kernels of the query language. Then, the query plan is developed for each query [2][3][4][5].

In this paper, we propose a query planning approach that can be used with column-store in-memory relational database accelerators. Our tool acts as a front-end compiler which takes a query at the input and generates a schedule that needs to be further processed to generate the instruction set of a P-DPU or the accelerator itself as a Q-DPU. It processes the SQL queries by using the given hardware abstraction of a P-DPU to generate the accelerator-optimized plan of a given query. If no abstraction is provided, it generates a generic plan column-store database processors. This plan can be used by the Q-DPU generating software. Within our knowledge, this is the first retargetable planner for column-store database processors.

Paper organization is as follows: A query plan is the ordered set of steps to retrieve and process data stored in a database. Traditionally, it is represented by a tree structure. In this paper, we enhance this tree representation by an acyclic synchronous data-flow graph (SDF) to model the streaming behaviour of the columns. Hence, next section defines SDF node types for SQL keywords. Then setting up the SDF representation of a query is described. In Section III, the hardware abstraction of the accelerator is mentioned. In Section IV, we explain our query planner. The final section concludes the work. In this paper, all columns that will be processed by the SQL query are assumed to be present in the memory.

II. SDF Vertex Types for SQL Operations

In a synchronous data-flow graph (SDF), vertices represent operations on the data, and edges represent queued or direct movement of data between operations. If there is no path between two vertices, then those operations can be executed in parallel. There can be different types of nodes with respect to their input and output tokens. The term "Input tokens" describes amount of data instances at the FIFO queues to fire a node (i.e., to execute an operation). At each execution of the node, a number of data instances, known as "output tokens" are generated. Let DI and DO denote the input and output tokens of a vertex, respectively. A node have multiple inputs and outputs. Let there be K inputs and L outputs of a node. Then we can define the following SDF node types for SQL expressions:

- **Combinational:** $DI_k = DO_l = 1$ for all $k \leq K$, $l \leq L$: A combinational node produces one set of output data for one set of input data. Logic operations and basic arithmetic operations are regarded as combinational nodes.

- **Semi-combinational:** $DI_k > 1$ for some $k \leq K$, $DO_l = 1$, for all $l \leq L$: Multiple instances of input data are required to produce a single data set at its output. Aggregate operations fall into this domain.
- **Synchronous:** $DI_k > 1$ for some $k \leq K$, $DO_l > 1$, for some $l \leq L$: Multiple instances of input data are required to produce multiple instances of data output. Data ordering and multiple table operations are synchronous operations.

Materialization is a concept to describe the stage when columns need to be retrieved from the memory. The types of the SDF nodes help us to roughly estimate the operation order of the nodes in the plan. For example, column filtering operations should be done as early as possible as long as I/O bandwidth, on-chip communication bandwidth, and on-chip resources of the accelerator are sufficient, because filtering (logic) operations are the fastest operations. In other words, early materialization should be supported in filtering operations if hardware architecture allows. For joins, our query planner should prefer late materialization because join is a computationally expensive operation. Hence, it should work on as small data as possible. Data ordering operations do not change amount of data. If they take place after joins, then sort time might be shorter because joins also reduce data. If they take place before joins, then the join time can get shorter because it is easier to join ordered tables. Hence, sort can take place before or after the joins. The place of arithmetic operations depends on the clause, which they appear in.

In SQL, a database table is created with CREATE TABLE that contains information about the data type of the columns. Data type provides the bandwidth information about a column. Hence, it is very useful during query planning for the hardware accelerator.

We form the SDF of an SQL query as follows: Columns from the tables appear as edges. Since all columns are coming from the tables, the START node holds the tables. For the START node, there is no input, i.e., $DI = 0$, but there are as many edges as the columns appearing in the SQL Query. Assuming that each column is read entry-by-entry, DO_l is 1 for each edge. Operations like filtering, order-by, join or group-by require one or more columns as keys. These operations affect all columns of the related table in the query. Hence all these columns are connected to the same node. In addition, the related SDF node has as many key ports as the key columns. Each key port is denoted with E on the node. Filter node of each table is separate. Columns are modified during query processing and represented with similar edge names. Columns that do not appear as outputs at the SELECT clause are discarded as soon as they are processed. While connecting an edge to a node, we use input and output tokens of the vertex. Data rates for the SDF vertices following a filtering operation are unknown because filtering operation will yield different results for different tables. For unknown amount of tokens, we use N. For example, $DI = N$, $DO = N$ for ORDER BY operation. This ambiguity simplifies design of the automated query planner as explained in Section IV. Based on the discussion on materialization, we propose SDF

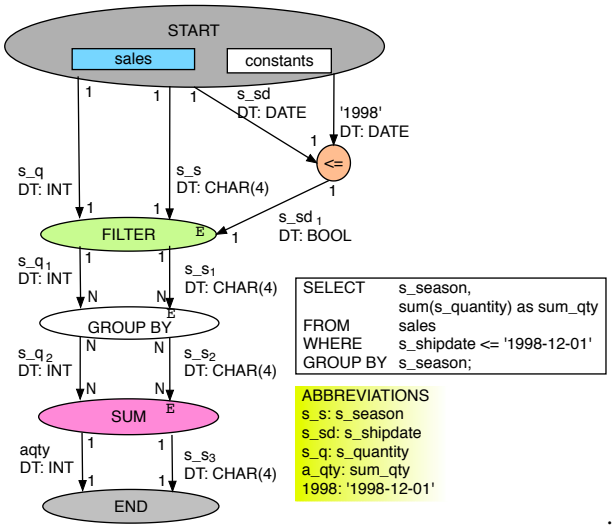

Fig. 1: An SQL query and its equivalent SDF

implementation order as FROM → WHERE → SELECT. If JOIN, GROUP BY and ORDER BY exist, then the process order is FROM → WHERE → JOIN → GROUP BY → SELECT → ORDER BY. In WHERE clause, SDF node for an equ-join (=) operation follows the filtering operations of the related columns. All outputs of the query are connected to an END node. Since, each output is produced entry-by-entry, DI_k of END is 1 for each column edge.

A simple query is shown in Figure 1. FROM clause contains only *sales* table. Hence, at the START node, the *sales* table is observed. In the query, *s_s*, *s_qty* and *s_sd* columns are used. Their data types are extracted from the *sales* CREATE TABLE. The constant values of the query appears in *constants* repository. Since WHERE clause has a comparison, a combinational node is placed for *s_sd* and '1998'. Its output is boolean which filters the remaining columns in the query. The FILTER node has an E port with s_sd_1 as a key. Similarly, both GROUP BY and SUM nodes contain an E port. Finally SELECT clause gives out the filtered version of *s_s*, namely, s_s_3 and aggregated *aqty* outputs. The SDF ends with the END node. Data types of all edges are specified by DT acronym in this figure.

III. Abstraction of a Hardware Accelerator

For designing a retargetable query planner, we need a hardware abstraction so that the query planner will be able to generate a valid plan for the related accelerator.

Our query planner requires the following information to be present in its execution environment: (1) Number of I/O and I/O bandwidth of the accelerator. (2) Communication network and bandwidth between modules. (3) Resource Allocation Table (RAT) for mapping each SQL statement to the correct hardware module. (4) Module Properties Table (MPT) which gives information about the type, amount, K (number of data inputs), L (number of data outputs), E (number of key inputs) DI for each input, DO for each output, latency and throughput

TABLE I: Resource Allocation Table (RAT) for AxleDB [5]

SQL statement	AxleDB Module
Arithmetic Expressions	{ARITH_32, ARITH_64}
(Logic Expressions, WHERE)	FILTER
Equ-join	(HASH_BUILD,HASH_PROBE)
SUM	AGGREGATE
JOIN	(HASH_BUILD,HASH_PROBE)
GROUP BY	HASH_BUILD
ORDER BY	SORT

TABLE II: Module Properties Table (MPT) for AxleDB [5]

	ARITH	FILTER	AGG	HASH BUILD	HASH PROBE	SORT
AMOUNT	16	16	16	1	1	1
K	12	12	12	12	12	12
L	12	12	12	12	12	12
E	4	4	4	4	4	4
DI_k	1	1	8	1	1	1
DO_l	1	1	1	1	1	128
Latency	1	1	3N	(1-16)N	(1-16)N	7N
Throughput	1	1	(1-3)N	(1-16)N	(1-16)N	7N

of each module on the accelerator. If not present, unlimited number of modules is assumed. (5) Data-type table (DTT) for identifying bit-length of each data type at the CREATE TABLE in the accelerator, because traditionally, bit-lengths of data types vary with respect to processing units. If not present, default values are used. (6) Connection matrix which shows the connection map between modules. If not present, full connectivity between modules is assumed.

For demonstrative purposes, we will use AxleDB Database Processor [5], which is a compile-time configurable processor targeting FPGAs. We picked a 32-bit configuration of the processor utilizing sixteen 32-bit input and sixteen 32-bit outputs simultaneously. A ring-network is used between the modules. 12 * 32-bit lines are reserved for data inputs and outputs, 4 * 32-bit lines are reserved for keys. Communication within the accelerator takes one unit cycle per one unit data transfer. Communication of the accelerator with the external memory or host also takes one unit cycle per I/O if sufficient bandwidth is provided. RAT and MPT tables shown in Table I and II can be extended for other SQL keywords. In RAT, we see that for Arithmetic Expressions, there are two ARITH units. The number after "_" shows that there are 32- and 64-bit versions that can be used with these expressions. The query planner selects the most suitable one according to the data type of the columns. If wordlength is not specified, it assumes 32-bits for this configuraton of AxleDB. The "()" shows the order of operations. For example, Logic Expressions are executed before WHERE in the F unit. In MPT, the terms DI and DO show the input and output data rates of the hardware module. For example, HASH BUILD (HB) and HASH PROBE (HP) modules work as soon as there is a single set of data at their inputs. This is different than AGGREGATE (AGG) unit because it has to wait until eight elements are ready at its inputs. Latency and throughput is given in terms of unit cycle. The term (1-16) means an operation may take 1 to 16 cycles. The term N shows that all columns need to be completely processed until valid results are obtained.

IV. Retargetable Query Planner

Query planning on SDF is a resource-constrained scheduling problem. While scheduling nodes with unknown data rates, the next step has to wait until the current step is finished. Let $\{u, v\}$ represent any two nodes in an SDF. Then "$DO_u = DI_v = 1$ for all edges between u and v" means direct data movement from one module to another module. In all other cases, a queue has to appear on each edge. This means that physical memory like FIFO or look-up table is required. If on-chip storage is not sufficient, then off-chip memory needs to be accessed.

To simplify the scheduling process, we revisit the techniques that were devised for known data rates. Since SDF of a query may contain nodes with nonuniform input and output tokens, we will partition the graph into Convex Connected Subgraphs of Uniform Frequency(CCSUF), which extends the Connected Subgraphs of Uniform Frequency (CSUF) terminology [12] with the convexity of a graph to provide the schedulability of the SDF:

Definition 4.1: Consider an SDF $G = \{V, A\}$, where V and A denote the set of vertices and edges of SDF respectively. Let $H = \{V_H, A_H\}$ denote the subgraph associated with $V_H \subset V$. Then H is a **Connected Subgraph of Uniform Frequency (CSUF)** if and only if H is a connected graph and for each edge $(u, v) \in A_H$, $DO_u = DI_v$.

Definition 4.2: Consider $H = \{V_H, A_H\}$, which is a CSUF of $G = \{V, A\}$. H is a **Convex CSUF (CCSUF)** if and only if there exists no path from a node $u \in V_H$ to another node $v \in V_H$ that involves a node $w \notin V_H$.

CCSUFs are formed starting from the END node because computationally expensive nodes are closer to the END. If an edge with $DI \neq DO$ is encountered, then CCSUF boundary of the current node is set, all nodes and edges of CCSUF are removed from the SDF. Then the process starts from the last node existing in the remaining SDF. Obviously, the resultant SDF might be disconnected. By applying the defined rules to Figure 1, two CCSUFs are obtained. Let them be named as CCSUF_1 and CCSUF_2. CCSUF_1 covers START, <=, FILTER nodes. CCSUF_2 covers the rest.

For each CCSUF, we apply bandwidth and resource-constrained scheduling algorithm by assuming that all hardware resources are available. In this paper, we extended the resource-constrained list scheduling algorithm to check for bandwidth requirements at each scheduling step. If internal bandwidth of the CCSUF exceeds I/O bandwidth, then maximum internal bandwidth is also reported as M. Then we apply ASAP scheduling algorithm to the reduced SDF by taking into account latency, resource utilization and I/O/M bandwidth of each CCSUF. If no resource/bandwidth violation occurs, we accept it as the query plan. Otherwise mobility-based scheduling applied with the latency of the ASAP scheduling. If still

resource violation continues, we firstly check mobility of each CCSUF. For the modules with nonzero mobility, we reduce the number of available resources and reschedule the graph of the CCSUF. The complete scheduling process is repeated until a query plan is generated with no resource/bandwidth violations. Then on each edge (u, v) where $DO_u > 1$ or $DI_v > 1$, queues are placed. In this paper, we used push-pop mechanism for its self-explanatory nature. For all the other edges, we use a "move" mechanism.

The query plan of our demonstrative example of Figure 1 is shown in Figure 2. We firstly schedule CCSUF_1. Since all edges in CCSUF_1 have $DO = DI = 1$, there are only move operations on these edges. Since AxleDB supports sixteen 32-bit inputs, then one "row" of all related columns can be moved in to the accelerator at the same control step (c_step 1). Here, "row" means one row of entry from *s_qty, s_s, s_sd* columns from *sales* table. At the same time, they can be fed to the FILTER block. Table II shows that there are sixteen filter modules in AxleDB. Since $DO = 1, DI = N$ from CCSUF_1 to CCSUF_2, FILTER output has to be pushed to queue QF whenever there is data on both s_qty_1, s_s_1. Then, scheduling of CCSUF_2 starts. Whenever there is data in QF, HB module pops and processes those columns. Note that it need not wait for the queue to be full, because MPT table shows that one set of inputs is sufficient to start. The output of this module has to be queued (QHB) since the edges in SDF between GROUP BY and SUM have $DO = N, DI = N$. In c_step 3, SUM operation starts on AGG module when QHB is not empty. Since AGG module of AxleDB takes eight inputs at a time, popping from QHB is done in chunks of eight, by using s_s_2 key. AGG unit produces two columns, which are *aqty,* s_s_3. In the last control step, these columns are moved to the output at each cycle. AGG does not wait all results to be completed to move the data to the accelerator output as DO of SUM in SDF and DO of AGG in MPT is 1. This means that the last values of $aqty$ for each s_s_3 are the correct values. In this planner, "c_step" is the control step where all sub-steps can occur in a pipelined manner, "row-by-row" of the stitched columns. c_steps can also be pipelined, because the related sub-step is taken only if the "condition" is satisfied.

To demonstrate query planning for limited bandwidth and resources, we can assume that there are only two 32-bit inputs and one FILTER . This splits the first control step into two control steps which filters and feeds *s_qty* and *s_s* into two queues. Note that this queue pair is actually QF of Figure 2. Yet, HB operation cannot start until there exists data in both queues. The rest of the query plan is the same with Figure 2 starting from c_step 3.

Some columns in a query may be wide enough to cover multiple inputs. In this case, a multicolumn appears in the query plan if and only if there is a supporting module in the hardware. When internal bandwidth is not sufficient, we regard it as the number of internal FIFOs is not sufficient, or vice versa. A similar case may also occur if two hardware modules are not directly connected in the accelerator but there exists a data transfer between the corresponding nodes at the SDF. In both cases, the query plan transfers the output of the first module to HOST via A_O and then reads-back by the accelerator via A_I whenever it is necessary.

c_step	CONDITION	u	v	(u, v)	KEY	IN	OUT
1	always	HOST	A_I	move	sales	s_qty s_s s_sd '1998'	
		A_I	F	move	$\{s\_sd \leq$ '1998'$\}$	s_qty s_s	s_qty_1 s_s_1
	when s_sd_1 = TRUE	F	QF	push		s_qty_1 s_s_1	
2	when QF $\neq \emptyset$	QF	HB	pop	s_s_1	s_qty_1 s_s_1	s_qty_2 s_s_2
		HB	QHB	push	s_s_2	s_qty_2 s_s_2	
3	when QHB $\neq \emptyset$	QHB	AGG	pop(8)	s_s_2	s_qty_2 s_s_2	$aqty$ s_s_3
4	always	AGG	A_O	move			$aqty$ s_s_3

F:FILTER, QF: Queue for F, HB: HASH BUILD, QHB: Queue for HB,AGG: AGGREGATE, A_I: Accelerator Input, A_O: Accelerator Output

Fig. 2: Query Plan of Figure 1 for AxleDB [5]

V. Conclusion

In this paper, we propose an SDF representation for column store databases. A tool that generates the SDF from the SQL query is designed. This tool currently handles all types non-hierarchical query plans. By using SDF representation with hardware abstraction tables, we have demonstrated generating a query plan specific to the accelerator.

Acknowledgment

The author thanks her former students, Kaan Bulut Tekelioğlu and Can Güler for the development of the software tool.

References

[1] M. Stonebraker et. al., "C-store: A column-oriented DBMS," *31st VLDB Conference*, 2005.

[2] J. Casper and K. Olukotun, "Hardware acceleration of database operations," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'14)*, pp.151-160, 2014.

[3] L. Wu et. al., "The Q100: Database Processing Unit," *IEEE Micro*, pp. 34-46, May-June 2015.

[4] S. L. Xi et. al., "Beyond the Wall: Near-Data Processing for Databases," *DaMoN'15*, pp. 2:1-2:10.

[5] B. Salami et. al., "AxleDB: A novel programmable query processing platform on FPGA," *Microprocessors and Microsystems*, pp. 142-164, June 2017.

[6] R. Mueller et. al., "Glacier: A Query-to-Hardware Compiler," *SIGMOD'10*, pp. 1159-1162.

[7] A. Becher et. al., "Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration," *FPL'14*, pp. 1-8.

[8] V. G. Castellana et. al., "High Level Synthesis of RDF Queries for Graph Analytics,"*ICCAD'15*,pp.323-330.

[9] Z. Wang et. al., "Relational query processing on OpenCL-based FPGAs," *FPL'16*, pp. 1-10.

[10] B. Sukhwani et. al., "Database analytics: a reconfigurable-computing approach", *IEEE Micro*, pp. 19-29, Jan.-Feb 2014.

[11] C. Pilato and F. Ferrandi, "Bambu: A Free Framework for the High-Level Synthesis of Complex Applications," *DATE'12*.

[12] S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping", *Journal of VLSI Signal Processing*, pp. 271-288, Dec. 1993.