AxleDB: A Novel Programmable Query Processing Platform on FPGA

BEHZAD SALAMI[1], Barcelona Supercomputing Center (BSC)
, Universitat Politcnica de Catalunya (UPC)- behzad.salami@bsc.es
GORKER ALP MALAZGIRT, Bogazici University- alp.malazgirt@boun.edu.tr
ORIOL ARCAS-ABELLA, Barcelona Supercomputing Center (BSC)
, Universitat Politcnica de Catalunya (UPC)- oriol.arcas@bsc.es
ARDA YURDAKUL, Bogazici University- yurdakul@boun.edu.tr
NEHIR SONMEZ, Barcelona Supercomputing Center (BSC)- nehir.sonmez@bsc.es

With the rise of Big Data, providing high-performance query processing capabilities through the acceleration of the database analytic has gained significant attention. Leveraging Field Programmable Gate Array (FPGA) technology, this approach can lead to clear benefits. In this work, we present the design and implementation of AxleDB: An FPGA-based platform that enables fast query processing for database systems by melding novel database-specific accelerators with commercial-off-the-shelf (COTS) storage using modern interfaces, in a novel, unified, and a programmable environment. AxleDB can perform a large subset of SQL queries through its set of instructions that can map compute-intensive database operations, such as filter, arithmetic, aggregate, group by, table join, or sort, on to the specialized high-throughput accelerators. To minimize the amount of SSD I/O operations required, AxleDB also supports hardware MinMax indexing for databases. We evaluated AxleDB with five decision support queries from the TPC-H benchmark suite and achieved a speedup from 1.8X to 34.2X and energy efficiency from 2.8X to 62.1X, in comparison to the state-of-the-art DBMS, i.e., PostgreSQL and MonetDB.

Additional Key Words and Phrases: Database Query Processing, Hardware Accelerators, Reconfigurable Computing

1. INTRODUCTION

In the rapidly growing field of Big Data, as the amount of data to manage increases, there is an increasing strain on Database Management Systems (DBMS) to meet high throughput and low latency requirements. The first main point of concern is the overhead of data movement that causes the throughput degradation, and decreasing the cache memory utilization per query. Secondly, conventional control-flow-based query processing engines can cause lower computational throughput compared to what can be achieved by application-specific hardware. From one side, to alleviate the overheads of data movement, one promising solution is to bring the computation closer to where the data resides, so that more operations can be completed avoiding non-essential data movement [1]. The gains are two-fold: easing the load on the host CPU for performing database operations, and reducing the negative impact on the performance of high-latency I/O operations. As a result, significant throughput improvements, as well as a reduction of I/O overheads can be achieved. On the other hand, streaming data through highly specialized hardware accelerators in a deeply pipelined fashion can significantly improve the computational throughput of the query processing engine.

FPGAs provide a unique opportunity to build an efficient query processing platform, by constructing a high-throughput execution engine with the additional aim of minimizing overheads of data movement. It is mainly the consequence of; ***(i)*** the inherent characteristics of massively parallel and configurable architecture of FPGAs, suitable for data streaming in deep pipelined-style execution ***(ii)*** the rise of High-Level Synthesis (HLS) technology, which makes FPGA applications relatively easier to develop compared to low-level languages such as VHDL or Verilog, and ***(iii)*** the availability of

[1]Behzad Salami is the corresponding author. Address: Nexus I Building, Office 303, Campus Nord UPC Gran Capita 2-4, 08034, Barcelona, Spain

soft cores that implement modern interfaces, such as PCIe 3.0 (Peripheral Component Interconnect Express) or SATA-3 (Serial AT Attachment).

For the query processing, FPGAs have been utilized in two distinct approaches: ***(i)*** traditional data offloading mechanisms, where data in the host-attached storage is offloaded towards external processing units or accelerators implemented in FPGA, or ***(ii)*** placing the processing units directly in the datapath between the host machine and the main storage units. The first approach could incur overheads stemming from the additional data movement since data needs to be offloaded through Operating System (OS) and device driver layers [2], [3], [4]. In contrast, the second approach allows processing units to get direct access to data blocks and facilitates low-latency data transmission [5], [6]. Also, it can correspond to significant speedups in the query execution.

By introducing a novel architecture for managing data movement through hardware accelerators and storage, in this work, we present the design and implementation of AxleDB. AxleDB is an FPGA-based engine that enables fast query processing by melding highly-efficient accelerators with commercial-off-the-shelf (COTS) storage using modern SATA-3, PCIe-3, DDR-3 (Double Data Rate) interfaces, in a novel, unified, and a programmable environment. Tightly coupled with a Solid State Disk (SSD) that stores blocks of database tables, AxleDB is designed to execute complex Structured Query Language (SQL) queries in full by performing various time-consuming query operations using specialized hardware accelerators, while the overhead of data movement between storage and compute units is also minimized. Thanks to the massively parallel and pipelined architecture of FPGAs and efficiently exploiting its embedded modules, i.e., low-latency on-chip RAM and DSP blocks, AxleDB provides a diverse set of high throughput query processing units for performing filtering, arithmetic, aggregation, group by aggregation, table join, and sorting operations. Also, to reduce I/O transfers, AxleDB supports FPGA-based database indexing, which is an efficient method to perform a quick scan of large database tables [7]. AxleDB is connected through a fast PCIe-3 interface and managed by a host system. In the host machine, PostgreSQL [8] runs as the host DBMS, which was enhanced to manage the data and instruction flow. AxleDB can process complex SQL queries using an extensive set of query-specific instructions, which can be generated from a given SQL query in the host.

In a nutshell, the main objectives of AxleDB as a novel query processing engine, are: ***(i)*** to provide an infrastructure that sits between host and data storage in SSD, and utilizes PCIe-3 and SATA-3 interfaces to work directly with blocks of database columns, ***(ii)*** to design a set of efficient query accelerators inside such an infrastructure that can facilitate the query processing in a fully pipelined fashion ***(iii)*** to investigate the efficiency of a database indexing method in the proposed FPGA-based platform as a technique to I/O transmitting reduction, and ***(iv)*** to allow the DBMS to utilize these accelerators by issuing AxleDB special instructions, exposing flexibility in data movement and in enabling accelerators. More specifically, the contributions of this paper are as follows:

— We describe the unified AxleDB platform that includes query processing-specific accelerators that are coupled to the storage device. Also, the efficient data management mechanism of the AxleDB to control the flow of data, which effectively enables rapid query processing in the hardware.
— We present novel and efficient database query processing accelerators for filtering, arithmetic, aggregation, groupby aggregation, table join, and sorting operations, as well as a MinMax database indexing method to reduce the disk I/O transfer. Leveraging them in a pipelined fashion leads to a high throughput query processing. To reduce the development time and to achieve more optimized designs we employed

modern HLS tools in various parts of the AxleDB. We explain our design decisions in detail while using Vivado HLS and Bluespec SystemVerilog.
— AxleDB is tailored for programmability and provides a diverse set of specific instructions for data movement and query execution. These query-specific instructions are generated at the host and used to program the AxleDB to process any given SQL query. For elaboration, we provide a step-by-step detailed query example from the commonly-used TPC-H benchmark.
— We evaluate the performance and energy efficiency of the AxleDB under various conditions, by running five decision-support TPC-H queries. We compare the AxleDB against the query processing engines of the state-of-the-art software-based DBMS, PostgreSQL, and MonetDB, in the single-threaded and multi-threaded modes. For the various class of the SQL queries, i.e., I/O-intensive, process-intensive, and I/O-process-balanced, we achieved speedups up to 34.2X and energy efficiency up to 62.1X, against the software DBMS.

The paper is organized into eight sections. Section 2 describes the architecture of the AxleDB by describing its major components. In Section 3 we illustrate the AxleDB by an example query to show how its components are utilized to process the example query. Section 4 introduces the supported database accelerators and the proposed database indexing method. The evaluation methodology is explained in Section 5. We discuss the experimental results in Section 6. Section 7 reviews the previous state-of-the-art query processing platforms, and finally, we conclude the paper in Section 8.

2. THE OVERALL ARCHITECTURE OF THE AXLEDB

In a relational DBMS, data is organized into tables, using a model of vertical columns and horizontal rows. The columns have a data type and a name. The rows represent entries in the database. The DBMS can store the data tables in two distinct models, row or column oriented storage. In column-oriented storage, all of the fields of a column are serialized and stored together. As opposed to row-oriented storage, DBMS can lead to higher I/O throughput as it would not need to load unnecessary columns of data in a column-oriented fashion. Exploiting SQL queries, meaningful information can be extracted by processing the data that is organized in tables. For this aim, the core query processing engine, as the backbone of a DBMS, manages the actual processing of SQL queries, following an established query plan. However, other functionalities of DBMS such as user authentication, logging, security, concurrency, etc., can be handled by other corresponding components rather than the core query processing engine. In this work, we concentrate on the query processing part itself, by attacking the computationally intensive query processing. We leave the other essential DBMS functionality for future work.

The main principle of the proposed database query processing platform, AxleDB, is to essentially move database computations closer to where the data resides, to obtain high performance in a flexible and programmable environment. Figure 1 shows the overall architecture of AxleDB. AxleDB resides between the host machine that runs the DBMS, and the database storage in an SSD. In the host, we primarily targeted to use PostgreSQL [8], one of the most popular open source relational DBMS. However, the infrastructure of AxleDB was designed to be software-agnostic and could be ported to other DBMS, e.g., MonetDB [9].

The host communicates with AxleDB through an Application Program Interface (API), to transfer data and instructions using the PCIe-3 interface. When the host initiates the query execution, the query plan needs to be converted into AxleDB instructions. Inside AxleDB, these instructions are managed and executed by the Data and

Process Controller (DPC), which orchestrates the movement of data blocks between SSD, DDR-3, host, and Accelerators. The query is effectively executed by streaming blocks of data, from the storage, through the accelerators, and back. Finally, the result of the query is returned to the software or stored back into the SSD. The architecture of AxleDB is composed of five major components:

(1) ***Software Extensions for DBMS in the host***, including the Data Address Table (DAT) and the CStore Foreign Data Wrapper (FDW) extension of PostgreSQL, to manage the transfer of instructions and data, respectively. We provide detail information in Section 2.1.1.
(2) ***Data Storage Units***, i.e., SSD, DDR-3, and host that are used as the primary or secondary database storage units and device controller cores, i.e., SATA-3, DDR-3, and PCIe-3 to manage the data transfer to/from storage units. We provide detail information in Section 2.1.2.
(3) ***AxleDB Accelerators Unit (AAU)***, which is a set of efficient DBMS query accelerators, i.e., filter, arithmetic, aggregation, group by, hash build, hash probe, and sort. To transfer data, accelerators are organized inside a ring bus, RingBus of AxleDB accelerators (RBAA), and a direct bus, DirectBus of AxleDB Accelerators (DBAA). We provide detail information about their overall interconnection in Section 2.1.3. Also, we introduce the internal structure of the accelerators in Section 4.
(4) ***Programmable Interconnection Unit (PIU)*** to set up a path to transmit the data in a fully flexible fashion. The PIU is composed of ***i)*** a 4-port bidirectional programmable data connection switch (PDCS) to exchange the data among SSD, DDR-3, host, and RBAA, ***ii)*** an arbiter to control the bandwidth sharing of DDR-3 by serializing its concurrent requests, and ***iii)*** a set of synchronizing First In First Out (FIFO) buffers for each individual port, separately for read and write directions, to cross the different clock domains. We provide detail information in Section 2.1.4.
(5) ***Data and Process Controller (DPC)*** that is composed of an Instruction Cache (IC) to locate the instruction set and an execute Finite State Machine (FSM) ***i)*** to manage the accesses to the off-chip data sources and ***ii)*** to control the accelerators to execute the corresponding query, by issuing the appropriate control signals to the PIU and to the AAU, respectively. These signals are generated by translating instructions of AxleDB. We provide detail information in Section 2.1.5.

We elaborate the aforementioned components in Section 2.1, individually.

2.1. Major Components of AxleDB

In this section, we elaborate the architecture of AxleDB and describe the role of each constituting component, individually.

2.1.1. DBMS Software Extensions for AxleDB. The host communicates with AxleDB for two purposes: ***i)*** to access the database tables that reside in the SSD, and ***ii)*** to program AxleDB using query-specific instructions in order to execute the SQL queries. In AxleDB, to process complex SQL queries, they first need to be translated to our specific instructions.[2] However, the certain currently unsupported operations, such as floating point division (DIV), can utilize a fallback-to-host scheme, and instead be executed in the software extension of AxleDB. We use the CStore FDW extension of PostgreSQL to access the database tables [10], which we refer to as 'CStore' for short. CStore manages data in a column-oriented format [11] that cause discarding unnecessary loads during the query processing and provides better I/O utilization. To transfer AxleDB

[2]The translation of SQL queries to AxleDB instructions is currently a manual process.

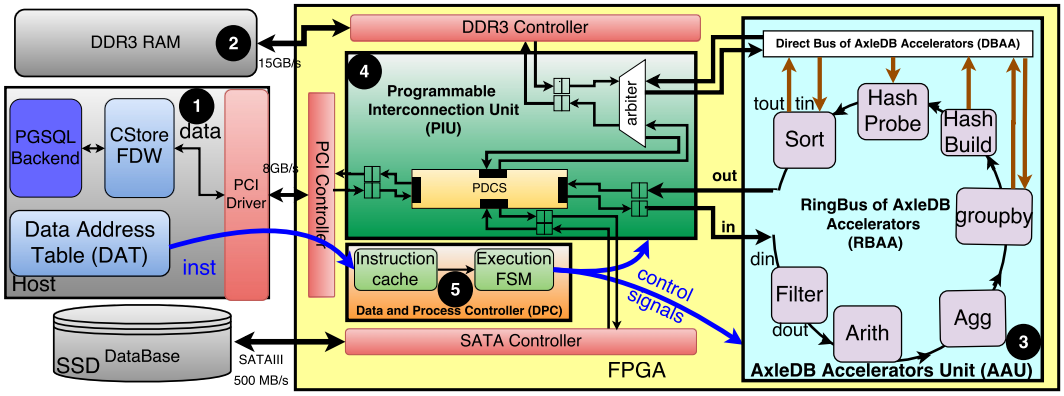


Fig. 1: Overall architecture of AxleDB with its major components. **(1)** software extensions for DBMS in the host, **(2)** Data storage units and device controllers, **(3)** a set of efficient query processing accelerators, which is so-called AxleDB Accelerators Unit (AAU), **(4)** Programmable Interconnection Unit (PIU) to manage the accesses to the off-chip data storage units, in a fully flexible fashion, **(5)** Data and Process Controller (DPC) to orchestrate the involved modules of AxleDB to process SQL queries.

instructions, we use a shared memory region between the host and FPGA, called Data Address Table (DAT). DAT resides in the host memory and holds the list of instructions of AxleDB and addresses of database tables. It is updated when the data tables are modified, or when a new query needs to be processed by AxleDB.

2.1.2. Data Storage Units and Device Controllers. AxleDB is connected to different sources of data, i.e., SSD, DDR-3, and host. ***i)*** As explained, the software extension of AxleDB is located on the host to initialize the configuration of the query execution, by issuing the query-specific AxleDB instructions. Its physical connection is through an PCIe-3 interface, with a maximum throughput of 8 GB/s. For this aim, we use Intellectual Property (IP) cores of Xilinx as the PCI-3 device controller. ***ii)*** SSD is connected through an SATA-3 interface, with a maximum throughput of 500 MB/s and is used as a primary storage to resides the database tables. Furthermore, we use a modified version of Groundhog [12] as the SATA-3 device controller. ***iii)*** DDR-3 is used as the secondary storage, with a maximum throughput of 15 GB/s. It is used to locate the input/output data tables and the temporary tables, e.g., hash tables, during the query processing. Also, we use the Memory Interface Generator (MIG) IP core of Xilinx as the DDR-3 device controller.

To reduce the overheads of data movement, in addition to the techniques as mentioned earlier, e.g., direct coupling of column-oriented SSD to the hardware accelerators, AxleDB is equipped with a database indexing method that is explained in Section 4.5.

2.1.3. AxleDB Accelerators Unit (AAU). In this section, we explain the overall organization and interconnection of AxleDB accelerators. AxleDB is equipped with a set of hardware accelerators to carry out the query processing primitives, i.e., filter, arithmetic, aggregation, groupby, hash build, hash probe, and sort units. Each accelerator has ***i)*** an input data port to stream in input data (*din*), ***ii)*** an output data port to stream out the result data (*dout*), ***iii)*** an input signal to determine the state of the unit (*state*), which is elaborated later in this section, and ***iv)*** a set of inputs to define the functionality of the given accelerator, e.g., to define the filtering qualifiers in the

filtering unit. Also, some of them have additional ports to access the temporary data during the processing, i.e., hash tables in the groupby, hash build, and hash probe units, and partially sorted data set in the sort unit. For this aim, ***i)*** the groupby unit has input/output ports (*tin*, *tout*) to read/write the hash table, ***ii)*** the hash build unit has an output port (*tout*) to write into the hash table, ***iii)*** the hash probe unit has an input port (*tin*) to read the hash table, and ***iv)*** the sort unit has input/output ports (*tin*, *tout*) to access the partially sorted data set. Other accelerators, i.e., filter, arithmetic, and aggregation units are inherently on-the-fly operations and do not need any access to the temporary data during the query processing.

To interface AAU with data storage units to transfer the input/output and temporary data set, the accelerators are connected through two distinct data buses, with respect to their sequential and random data access types. More specifically, ***i)*** the input/output data are usually accessed sequentially. Thus, the potential long latencies can be covered by streaming data in a pipelined schema. Accordingly, to access the input/output data, our design decision is made due to providing a flexible schema. In contrast, ***ii)***to access the temporary data, we set a shortcut path to DDR-3 with a minimum latency. Since the temporary data, i.e., hash table and partially sorted data, are accessed randomly, thus, a low-latency path would be efficient. We elaborate the interconnections, as below:

(1) ***RingBus of AxleDB Accelerators (RBAA):*** To build AxleDB flexible enough, we organize the accelerators inside a unidirectional ring structure that is so-called RBAA. We use the RBAA to only stream in/out the input/output data tables, and not temporary data, from/to the data storage units, i.e., SSD, DDR-3, and host, in a flexible schema. As it can be seen in Figure 1, the accelerators are chained to each other with a specific order, as follows: filter, arithmetic, aggregation, groupby, hash build, hash probe, and sort units. To set up the chain, we connect the *dout* port of the earlier unit to the *din* port of the latter unit, consecutively. Also, the *din* port of the filter unit and the *dout* port of the sort unit are used to externally interface the RBAA. Also, currently, we design RBAA as a single-channel bus, which leads to a single stream of data in the ring, at a time. Accordingly, to process an SQL query, we need to break it into a set of data streaming paths. By streaming data through the data paths, in a sequential order, as RBAA is a single-channel bus, processing of the corresponding SQL query can be accomplished. We will elaborate this process in Section 2.2 and also illustrate it for an example query in Section 3.

(2) ***DirectBus of AxleDB Accelerators (DBAA):*** As mentioned earlier, groupby, hash build, hash probe, and sort units needs to be connected to an off-chip storage to access their temporary data set. For this aim, we use a dedicated data bus that is so-called DBAA. As it can be seen in Figure 1, the *tin* and/or *tout* ports of accelerators are connected to this data bus. It is worth noting that accessing random data set, e.g., hash tables and partially sorted data set, under a long latency would cause a significant throughput degradation. Although, some techniques such as multithreading [22] can alleviate this issue, in the current version of AxleDB, our sort and hash-based accelerators are single-threaded. Alternatively, in AxleDB, we make some design decisions to decrease the latency of accessing the temporary data, by dedicating a direct data bus, as:
   — We believe that DDR-3 is the only promising accommodation among the available data storage units, i.e., SSD, DDR-3, and host, to cope with the temporary data. Since, DDR-3 has the shortest latency, which qualifies it for random data accesses such as hash tables and partially sorted data set. Thus, an interconnection to DDR-3 is designed for providing quick temporary data access of the aforementioned accelerators.

— Accessing DDR-3 through the ring bus incurs an additional latency (at max 7 cycles for each data access), which as explained, may cause performance degradation. Thus, to access the temporary data, we dedicate the direct bus, so-called DBAA without any additional latency. Similar to the RBAA, the DBAA is also a single-channel bus, which corresponds to transfer a single set of data, at a time.

Also, as it is explained in Section 4.3 and 4.4, we substantially optimize the architecture of the corresponding hardware accelerators by using the on-chip RAMs of FPGA to discard the latency of the temporary data access during the query processing.

In summary, we equip AxleDB with ***i)*** RBAA to provide flexibility to access the input and output data tables from various data storage units, and ***ii)*** DBAA to quick access to the DDR-3 for some of the costly query operations, i.e., group by, hash build, hash probe, and sort units. The inherent advantage of having two distinct data buses is maximizing the overall throughput when various storage units are exploited for each data bus. For instance, by using SSD for the RBAA and DDR-3 for the DBAA, the bandwidth of both SSD and DDR-3, can be utilized, at a time. On the other hand, assuming that the RBAA is also connected to the DDR-3, the bandwidth of the memory needs to be shared among RBAA and DBAA. To share the bandwidth of DDR-3, we would need an arbiter that serializes the concurrent DDR-3 requests. The arbiter is a part of PIU, which we give more details on Section 2.1.4.

Nevertheless, in accordance to the SQL queries, a subset of the hardware accelerators are usually utilized at a time and not all of them. To meet this requirement in the hardwired chain structure of the RBAA, we design the accelerators to work in two distinct states: *active* or *silent*, which can be set by using an input signal. In the *active* state, the accelerators normally work, as expected to carry out the expected function of the query primitive, e.g., filtering, aggregating. In contrast, in the *silent* state, the accelerators work as bypass buffers and only pass the incoming data to the next unit in the ring (with a single-cycle latency), without applying any function. As a consequence of this structure, depending on the SQL queries, we can utilize only the required subset of the accelerators, by setting them to the *active* state and by setting other accelerators to the *silent* state. Considering the architecture, to access data, the chain incurs a latency of maximum 7 cycles. However, as mentioned earlier, we use it only for streaming input and output data tables and not for temporary data, which does not cause any throughput degradation. Because the additional latency is covered by streaming the sequential data access.

Note that the structure of the AAU may constrain supporting the SQL queries or achieving a fully optimal query plan of a given query. Below, we discuss the possible constrains of AxleDB:

(1) ***RBAA as a Hardwired Chain of the Accelerators:*** Although, the accelerators are chained inside a hardwired ring, the current order is viable enough to make an efficient query plan. Performing a filtering operation at the beginning can significantly reduces the size of the data for the next costly operations, i.e., group by, hash build, hash probe, and sort units, as they need frequent accesses to the off-chip DDR-3. In summary, by following this design point, the off-chip data accesses can be decreased, which in turn, corresponds to a significant throughput. We show more detail on this design point in Section 3.3 for an example query.

(2) ***DBAA as a Single-channel Data Bus:*** In the DBBA, only a single stream of data can use the bus, at a time. In other words, at a time, one of the corresponding accelerators, i.e., group by, hash build, hash probe, and sort, can be active. Consequently, in an AxleDB-specific query plan for a given SQL query, we need to be ensured that each sub-query exploits only one of the aforementioned hardware units to access

random temporary data in DDR-3. We elaborate the query planning of AxleDB for an example query in Section 3.

However, this is not an optimal architecture, but it works for running complex queries in a fast and flexible way. Also, for any unsupported SQL queries, we can either ***i)*** if possible, rewrite the SQL query toward an equivalent and a more adapted version with AxleDB or ***ii)*** call the software extension of AxleDB to process the query.

2.1.4. Programmable Interconnection Unit (PIU). PIU is designed to make AxleDB flexible enough to exchange the data among SSD, DDR-3, host, and RBAA. Also, it synchronizes the data movement among different clock domains and manages the bandwidth sharing. The PIU is composed of the following components:

— A 4-port Programmable Data Connection Switch (PDCS) to exchange the data blocks among SSD, DDR-3, host, and RBAA. To support all the possible data movement cases among the data sources, its ports are designed to be bidirectional, whereas each direction of each port can be utilized independently.
— The buffering and synchronizing FIFO modules to manage the data movement among the various ports of the PDCS. For the data transfer, buffering and synchronization is a crucial mechanism because, first, various data sources have different latencies, and second, they work with different clock frequencies. For each read and write direction, separate FIFOs are dedicated.
— An arbiter to manage the DDR-3 read/write requests, as DDR-3 has two distinct access modes, first, a direct connection from some of the accelerators in the DBAA to access their temporary data and second, an indirect connection through the PDCS to access the input/out data tables. In the arbiter, among the aforementioned concurrent DDR-3 requests, we set the higher priority to the requests from direct connection DBAA to quickly serve the requests for the temporary data.

In summary, PIU is designed to efficiently share the bandwidth of the data storage units, while it can provide a fully flexible data movement schema.

2.1.5. Data and Process Controller (DPC) and AxleDB Instruction Set. DPC orchestrates the involved modules of AxleDB to process the complex SQL queries in a fast, efficient, and flexible schema. More specifically, it manages the movement of data and the activation of hardware accelerators to execute an SQL query, by issuing the control signals to the PIU and the AAU, respectively. Accordingly, the control signals from DPC, ***i)*** to the PIU determine the source and the destination of the data movement, and ***ii)*** to the AAU activate those accelerators that need to be utilized in the RBAA for any certain SQL query. Also, the parameters of each accelerator are determined by DPC, e.g., the filtering qualifiers for the filter unit.

The control signals are used by DPC to manage the query processing. They are generated based on instructions of AxleDB. The query-specific AxleDB instructions are translated from the SQL queries in the host. Translation of the SQL queries to the instruction set of AxleDB is currently a manual process. We introduce instructions of AxleDB later in this section. As it can be seen in Figure 1, instructions are located in a shared location in the host memory that is called DAT. Then, they are sent to the IC that resides inside the DPC. The DPC fetches them from the IC, decodes, and executes by the execution FSM. Consequently, the control signals are issued and sent to the PIU and the AAU. Accordingly, the query execution is completed, when all the instructions inside the IC are consumed. In the end, the results of the query can be either stored back in the SSD or sent to the host. The DPC can now synchronize with the host and wait for more instructions to process a new query.

Table I: AxleDB instructions for data movement and query execution.

Data Movement Instructions (DMI)		**Query Execution Instructions (QEI)**	
Instruction	**Description**	**Instruction**	**Description**
HOST-SSD#	streaming data between host and SSD	filter#	a generic filtering
		arith#	an arithmetic op.
HOST-DDR#	streaming data between host and DDR	aggregate#	an aggregation op.
		groupby#	hash-based groupby
SSD-DDR#	streaming data between SSD and DDR	hash_build#	building the hash table
		hash_probe#	probing the hash table
HOST-DCU#	sending instructions from host to DCU	sort#	sorting data
		minmax_index#	index checking

Table I summarizes the instruction set of AxleDB for performing the tasks of data movement and query execution. Data Movement Instructions (DMI) set up the PDCS to exchange data blocks among the different sources of AxleDB, i.e., SSD, DDR-3, and host, bidirectionally. Furthermore, depending on the query plan, Query Execution Instructions (QEI) configure AxleDB to activate the corresponding accelerators in the RBAA to start streaming the input data. QEI consist of filtering, arithmetic, aggregation, group by, hash probe, hash build, and sort instructions, as well as MinMax index creation/deletion instructions. Following by this way, AxleDB supports a large subset of the SQL queries, by mapping them to the architecture of AxleDB, which provided by using AxleDB instruction set. However, as mentioned earlier, the unsupported operations can be fallback to the software extension of AxleDB to accomplish the query processing. DMI and QEI include parameters, e.g., source, destination, key columns, carried column (payload), as well as accelerator-specific parameters, e.g., filtering operations ($<, >, <>$) and its qualifiers for the filtering unit.

2.2. The Execution Model of AxleDB

To alleviate the common restrictions of classical processor-based systems, the execution model of AxleDB relies on the streaming of the data through the processing units. For this aim, FPGAs provide a unique opportunity, whereas their programmable logic blocks that are called LookUp Tables (LUT) can be chained together to construct deep pipelines. In this model, each processing node in the pipeline can be enabled, whenever the inputs are available.

To process an individual SQL query, we use AxleDB instructions to establish the required data streaming paths. In other words, in AxleDB, each SQL query is defined by a set of the data streaming paths. Source and destination of data, e.g., SSD, DDR-3, host, together with the required processing units in the AAU constitute a data streaming path. Accordingly, to process an SQL query, we need to make a query plan by breaking the SQL query to a set of sub-queries, considering the constraints of AxleDB's structure that is discussed in Section 2.1.3. The generated sub-queries are one-by-one mapped to data streaming paths that can be run in AxleDB. Later on, we generate the required instructions and configure AxleDB to establish the corresponding data paths, sequentially. And finally, by streaming the data through the established data paths, the processing of the query can be accomplished. In the current version of AxleDB, at a time, we can establish a single data streaming path. This property leads to a sequential, in order and one by one, execution model for the data streaming paths, which lead to having single stream of data in components of AxleDB, at a time.[3] Accordingly,

[3]However, we believe that by adding parallel channels of data sources in the second version of AxleDB, it can also support the parallel data streaming paths. This is an ongoing work.

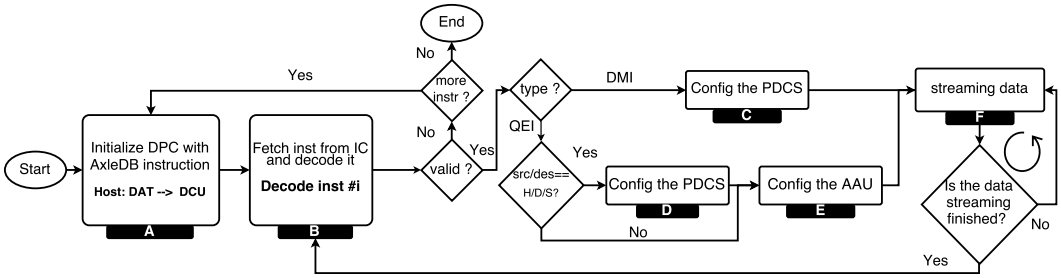


Fig. 2: The flowchart of the execution model in AxleDB. (In this figure: H= Host, S= SSD, and D= DDR-3.)

we elaborate the process of defining the data streaming paths by using instructions of AxleDB, i.e. QEI, DMI. The simplified flowchart of the execution model is shown in Figure 2. As it can be seen, first, the DPC is initialized by a set of AxleDB instructions that are copied from DAT in the host to the IC in the FPGA (A). Depending on the type of each valid instruction (B):

— DMI lead to exchange data among SSD, DDR-3, and host. Thus, after appropriately configuring the PDCS to set up the required source, destination, and direction, (C) the data are streamed in (F).
— QEI imply that data needs to be processed in the hardware accelerators, which lead to access the AAU. In the AAU, the corresponding accelerators are activated, and others are configured only to pass the data (E). Furthermore, for those QEI that need to transmit data to/from SSD, DDR-3, or host, which is determined by the parameters of the instructions, configuring the PDCS is also needed (D). Finally, data streaming is started through the established data path. As mentioned, at a time, we have a single stream of data in components of AxleDB. Thus, before reading new instruction, the current stream needs to terminate executing (F). Later on, we proceed to read the next instruction from IC (B) to start making the next data streaming path.

This process continues until consuming all the instructions of the IC while updating DAT with new instructions can restart the execution process of AxleDB.

3. ILLUSTRATING THE EXECUTION MODEL OF AXLEDB BY AN EXAMPLE QUERY

In this section, to demonstrate how AxleDB works, we illustrate the query processing procedure for an example query. The example query is Q03 from TPC-H benchmark [13]. AxleDB runs this query without any required modifications or code rewriting, where most of the accelerators presented in this work, are utilized. Since AxleDB is designed to be programmable, we can follow many different query plans to execute the queries. However, in this example, to show a comprehensive execution model of AxleDB, where input data is located in the SSD, we built a customized query plan. Accordingly, we first, load the input data from SSD to the DDR-3, and then, start query execution by retrieving data from DDR-3. In the rest of this section, we first, introduce the example query. Later on, show how AxleDB processes this example query by describing an optimal query plan, by introducing the list of the required AxleDB instructions to run the query plan, and by explaining how these instructions program the components of AxleDB to utilize the required modules.

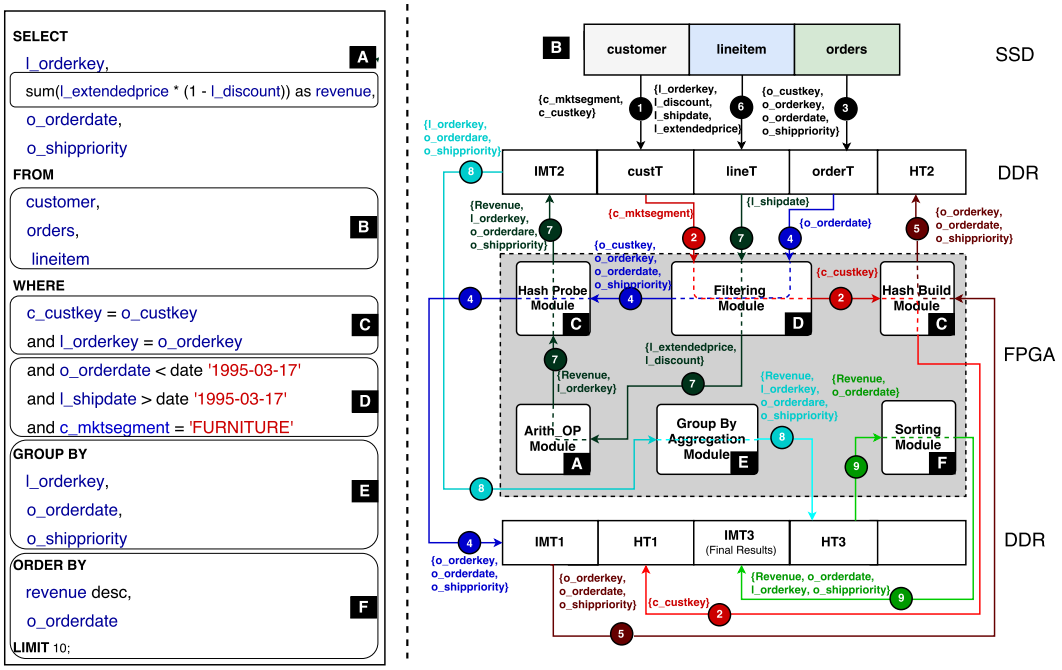


Fig. 3: (a) Example SQL query: Q03, (b) An example query plan of AxleDB to process Q03. To have a simpler figure, ***i)*** we partition the tables of the DDR-3 memory into two boxes at above and below of the FPGA, although, AxleDB is currently attached to a single channel of DDR-3, and ***ii)*** we only show essential fields of the labels of arrows, excluding the input parameters of accelerators, payloads, etc.

3.1. Elaborating the Example Query

The example query is shown in the Figure 3 (a). In a typical SQL query, several language elements such as **SELECT FROM**, **WHERE**, **GROUP BY**, and **ORDER BY** can exist. These operations can be semantically mapped to specialized hardware accelerators. In this example, the **SELECT** statement fetches the desired data columns (*l_orderkey*, *revenue*, *o_orderdate* and *o_shippriority*) **FROM** the given tables (*customer*, *orders* and *lineitem*). The **WHERE** statement is used to restrict the data in the tables and includes operations such as logical comparisons and arithmetic operations that filter the data relevant to the user (e.g. *o_orderdate* $<$ date *1995-03-17*, *l_shipdate* $>$ date *1995-03-17* and *c_mktsegment* = *FURNITURE*). AxleDBs filtering units can be exploited to handle the **WHERE** clauses of SQL queries. When multiple tables are involved, the join statement (*c_custkey* = *o_custkey* and *l_orderkey* = *o_orderkey*) is used to combine the data in these tables, based on a common field (*custkey* and *orderkey*). This operation can be efficiently mapped to AxleDBs hash join accelerator. The **GROUP BY** statement aggregates data into groups based on a given field (i.e. *l_orderkey, o_orderdate, o_shippriority*), which is mapped to AxleDBs hash-based groupby accelerator. The **ORDER BY** statement sorts the data in ascending or descending order based on a given key (i.e. *o_orderdate, revenue*), which can be performed using AxleDBs sorter accelerator. The **LIMIT** statement causes to fetch a limited number of records. As it is further explained in Section 4.4, this operation is currently melded inside of AxleDBs sorter module.

3.2. How does AxleDB Process the Example Query?

To process an SQL query in AxleDB; first, the host generates a set of DMI and QEI. Currently, this is a manual process, but it can be automated by following a similar approach with Glacier [14]. Figure 3 (b) shows a simplified diagram for one possible query plan for processing the query Q03 on AxleDB, where the mapping from the key operations of the query to AxleDBs accelerators is indicated by letters (from A to F). To have a simpler figure, we did not show many details of AxleDB, i.e., RBAA, PDCS, instruction parameters, etc. Also, as it can be seen, the execution is accomplished in nine distinct steps. Each step is distinguished by using a set of arrows with its unique numbers and colors in the figure. The label of each arrow shows the key columns that are used in the corresponding part of the processing. Within each step, data can be streamed in a pipelined fashion, and between steps, it is sequential (in order and one by one), due to data dependencies.

Due to column-oriented data store format of AxleDB, only the 10 required columns out of a total of 33 columns (of three input data tables) are loaded from SSD to AxleDB. On the other hand, while loading data from SSD to DDR-3 memory, the columns of data are converted into batches that are appropriate for the processing units. Also, during the processing, different types of data can be stored in the DDR-3 memory, i.e., input data tables (custT, ordersT, lineT), intermediate data tables (IMT1, IMT2, IMT3), and temporary data tables, e.g., hash tables (HT1, HT2, and HT3). To access to the aforementioned data tables from the DDR-3, the memory bandwidth is shared. As explained in Section 2.1.4, we manage bandwidth sharing by using an arbiter to serialize the concurrent memory requests. We elaborate it in Section 3.3 for a sample data path, step #7 as below, of the example query. In a nutshell, we perform the following steps to run Q03 on AxleDB (The presented numbers are for the 1GB scale of the dataset. However, more information about our benchmark environment is presented in Section 5.3.):

(1) Query processing starts by loading only the necessary columns of *customer* table to DDR-3, using 'DMI: SSD-DDR#'. *c_mktsegment* and *c_custkey* columns are loaded to DDR-3 and others are skipped.
(2) For *customer* table, first performing a filter on *c_mktsegment*, using 'QEI: filter#' reduces size of data from $\approx$ 150K to $\approx$ 30K records. Later on, for the filtered data, a hash table (HT1) is built into the DDR-3 based on *c_custkey* field, using 'QEI: hash_build#'.
(3) Query processing resumes by loading only the necessary columns of *orders* table to DDR-3, using 'DMI: SSD-DDR#'. *o_custkey*, *o_orderkey*, *o_orderdate* and *o_shippriority* columns are loaded to DDR-3, and the others are skipped
(4) For *orders* table, first performing a filter on *o_orderdate*, using 'QEI: filter#', reduces the size of dataset from $\approx$ 1.5M to $\approx$ 725K records. Later on, HT1 is looked up based on *c_custkey*, using 'QEI: hash_probe#'. The resulting joint table is stored into the DDR-3 (IMT1) with $\approx$ 145K records of data.
(5) Applying a nested hash join process, IMT1 is used as the input table to build the second hash table based on *o_orderkey*, using 'QEI: hash_build#'. The hash table is stored into HT2.
(6) Query processing continues by loading the necessary columns of *lineitem* table, using 'DMI: SSD-DDR#'. *l_orderkey*, *l_extendedprice*, *l_doscount* and *l_shipdate* columns are loaded to DDR-3, and others are skipped.
(7) For *lineitem* table, first performing a filter on *l_shipdate*, using 'QEI: filter#', reduces the size of the input dataset from $\approx$ 6M to $\approx$ 3.2M records. Later on, in a pipelined fashion the arithmetic unit is exploited to compute *revenue*, using 'QEI: arith#'. Probing the second hash table (HT2) based on *l_orderke*, using QEI:

hash_probe#, it generates the resulting joint table into IMT2, with about 30K records of data.

(8) In this step, data records of IMT2 are grouped into the new table (HT3), based on a merged *key* (*o_orderkey*, *o_orderdate*, *o_shippriority*). In addition, an aggregation on *revenue* field is performed, using 'QEI: groupby#'. A total number of groups (records) is $\approx$ 11K in HT3.

(9) The query processing is finalized by sorting all groups of HT3 based on a merged *key* (*revenue*, *o_orderdate*), using 'QEI: sort#', and transferring top 10 records to PostgreSQL, using 'DMI: DDR-HOST#'. (The host is not shown in the diagram, for a clearer figure.) The final result in IMT3 can also be written into the SSD, using 'DMI:SSD-DDR#'.

Due to explained query plan of Q03, the required instruction set of AxleDB to process the given query is summarized in Table II. They are composed of many parameters, e.g., the operation (that defines the appropriate operation), the source (to determine the source of the data and corresponding address), the destination (to determine the destination of the data and the corresponding address), the key columns (the columns that are used as key during the query execution) and the payload (the columns that are only carried along). In summary, these instructions are used to establish the required data streaming paths in AxleDB to process the example query Q03, by following the query plan in Figure 3. Accordingly, as an example, in Section 3.3 we illustrate how these instructions are used to establish one of the sample data streaming paths, #7, by following the process in Figure 2.

3.3. Establishing a Data Streaming Path: Elaboration for a Sample Data Path

In this section, we explain how components of AxleDB are leveraged to process the example query. As it can be seen in Table II, the processing of Q03 can be accomplished by 13 AxleDB instructions that lead to creating 9 distinct data streaming paths. Among them, and as an example, we explain the required steps to create the sample data streaming path #7, which is depicted in Figure 4. As it can be seen, to establish the given data path #7, 3 QEI are used (#9, #10, and #11). Each QEI determines a specific part of the data path #7 and finally, by the last instruction the path is established. Due to each instruction, the DPC generates the necessary control signals to the PIU, to set up the data movement path, and to the AAU, to activate and configure the required hardware accelerators. Accordingly, to establish the given data path, steps as below are proceeded:

(1) ***Instruction #9:*** As it can be seen in Figure 4(a), instruction #9 defines a filtering operation for the data in the DDR-3. Thus, the required actions are ***i)*** configuring the PDCS to stream in the data from the DDR-3, in this case lineT, to the AAU-RBAA. For this aim, the appropriate ports of the PDCS are utilized. And, ***ii)*** activating the corresponding hardware accelerator, in this case filter unit, by setting it to work in the *active* state (state=1). Also, the query-specific filtering parameters are defined by DPC, "*l_shipdate* $>$ *1995-03-10*".

(2) ***Instruction #10:*** As it can be seen in Figure 4(b), instruction #10 defines an arithmetic operation for the filtered data. Thus, the only required action is to activate the arithmetic accelerator by setting it to work in the *active* state (state=1). Also, the convenient parameters of the arithmetic unit need to be set. For this aim, the DPC generates the required control signals to carry out the corresponding arithmetic operation, "*l_extendedprice*(1-l_discount)*". It is worth noting that for this instruction, as it does not have any valid source or destination parameters, we do not need to modify the PDCS configuration.

Table II: AxleDB instructions to process the example query, according to the query plan in Figure 3 (some of the fields such accelerator-specific parameters are omitted in this table.) The size column shows the size of the data stream for each data path, in terms of the number of rows, in 1 GB scale dataset.

#path	**#instr**	**Instruction Fields**					
		operation	**src**	**dest**	**key_cols**	**payload**	**size (1GB)**
1	1	SSD-DDR	customer (SSD)	custT (DDR)	c_mksegment, c_custkey	—-	150K
2	2	filter	custT (DDR)	—	c_mksegment	c_custkey	
	3	hash_build	—	HT1 (DDR)	c_custkey	—-	30K
3	4	SSD-DDR	orders (SSD)	orderT (DDR)	o_custkey, o_orderkey, o_orderdate, o_shippriority	—-	1.5M
4	5	filter	orderT (DDR)	—	o_orderdate	o_custkey, o_orderkey, o_shippriority	
	6	hash_probe	—	IMT1 (DDR)	o_custkey	o_orderkey, o_orderdate, o_shippriority	0.72M
5	7	hash_build	IMT1 (DDR)	HT2 (DDR)	o_orderkey	o_orderdate, o_shippriority	0.14M
6	8	SSD-DDR	lineitem (SSD)	lineT (DDR)	l_orderkey, l_discount, l_shipdate, l_extendedproce	—-	6M
7	9	filter	lineT (DDR)	—	l_shipdate	l_orderkey, l_discount, l_extendedproce	
	10	arith	—	—	l_extendedprice, l_discount	l_orderkey	30K
	11	hash_probe	—	IMT2 (DDR)	l_orderkey	Revenue, o_orderdate, o_shippriority	
8	12	groupby	IMT2 (DDR)	HT3 (DDR)	l_orderkey, o_orderdata, o_shippriority	Revenue	11K
9	13	sort	HT3 (DDR)	IMT3 (DDR)	revenue, o_orderdate	l_orderkey, o_shippriority	11K

(3) ***Instruction #11:*** As it can be seen in Figure 4(c), instruction #11 completes the establishing of the data path #7, as it needs to write data into one of the data sources, DDR-3. Thus, data streaming will be started after this final step. Instruction #11 requires utilizing the hash probe accelerator by setting its state to *active* (state=1). Consequently, the other accelerators in the AAU including aggregation, group by, hash build, and sort units are configured to work in the *silent* state to only pass the incoming stream of data to the next unit in the RBAA. Also, the hash probe key, in

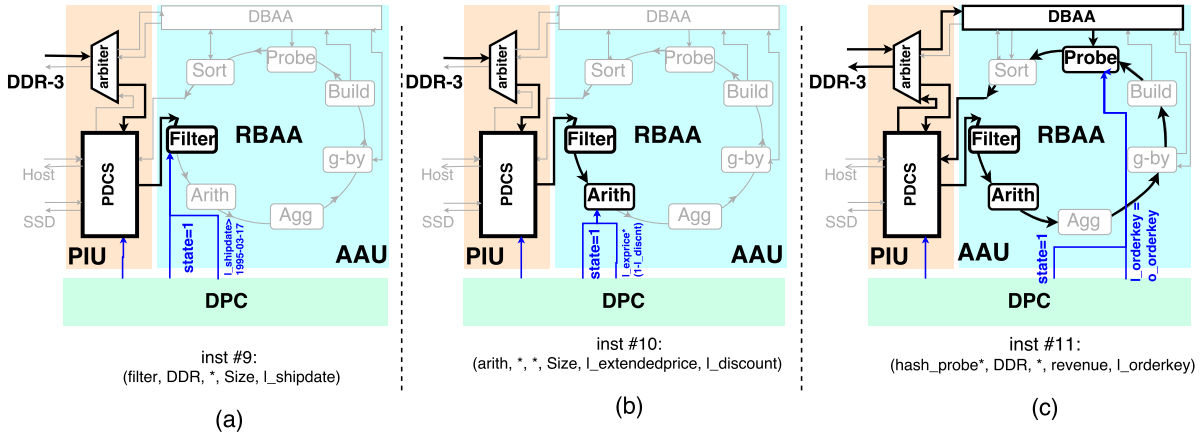


Fig. 4: Establishing the data streaming path #7 by composing together different AxleDB instructions #9, #10, and #11. The detailed connections between DPC, AAU, and PIU are shown. Among them, the highlighted components/connections represent the corresponding parts that are utilized by each AxleDB instruction to set up the given data streaming path. The control signals for the PIU and the AAU are generated by DPC.

this case *l_orderkey*, is determined by the DPC. The hash probe unit needs another DDR-3 memory access to read the hash table, in this case HT2. For this aim, the corresponding port of the DBAA is activated to access the hash table from DDR-3.

In summary, the data streaming path #7 leads to ***i)*** read the lineT table from DDR-3, ***ii)*** filter the incoming data stream based on *l_shipdate* item, ***iii)*** apply an arithmetic function for the filtered data, ***iv)*** probe the stream of the data based on the *l_orderkey* in the HT2 hash table., and finally, ***v)*** write the probed data into the DDR-3 in the index IMT2. In total, there are 3 distinct DDR-3 data access paths in this data path, i.e., ***i)*** to stream in the input data (as explained above in the part of instruction #9) through RBAA, ***ii)*** to access the hash table in the hash probe unit (as explained above in the part of instruction #11) through a direct DBAA bus, and finally, ***iii)*** to stream out the result data (as explained above in the part of instruction #11) through RBAA. As mentioned in Section 2, to manage the concurrent DDR-3 requests and optimally share its bandwidth, the arbiter in the PIU is exploited, which allows the priority to the DBAA requests to quickly serve the hash table accesses.

4. QUERY PROCESSING ACCELERATORS

In this section, we go through the architecture of the proposed accelerators that are implemented to perform efficient query processing in AxleDB. We proposed query execution units including filtering, arithmetic, aggregation, sorting, hash join, and groupby, as well as the MinMax indexing mechanism for I/O optimization.[4]

For developing, leveraging HLS tools, we have designed the filtering and aggregation accelerators in Vivado HLS [15], where we have been able to exploit the data parallelism via Vivados compiler directives. For task-parallel and control-oriented accelerators, such as the hash join and sort engines, Bluespec SystemVerilog [16] is used.

[4]In the rest of the paper, we assumed input data table as a set of tuples, pairs of *key* and *value*. *key* refers to the column(s) of data, used for performing the main query operation, e.g., sorting key in a sorter unit. *Value* refers to the other columns that need to be carried to make the final resulting data.

Verilog RTL code is employed for the integration of interface controllers. We select to employ these HLS tools as a result of our previous empirical analysis of a representative set of HLS tools for database acceleration [17].

4.1. Filtering Operations, Arithmetic, and Logic Unit

Database filtering is relational operations that test the numerical or logical relations between columns, in the form of numerical and/or Boolean values. The key importance of filtering operations in an SQL query is to reduce the amount of data for further processing [18], [19]. For this purpose, we designed a compile-time parameterizable, variable width, n-way compute engine that takes in rows of data as inputs, applies a filtering operation to the desired fields and produces an output bitmap. This bitmap determines the resulting rows for further processing. Similarly, we designed arithmetic, and logical compute engines. The arithmetic engine supports the integer Add, Sub, and Mult operations[5], whereas the logical engine supports the NOT, AND, OR, NOR and NAND operations. Also, keywords such as IN, SOME, and EXISTS can also be mapped to multiple logical operations. The design behind the filtering, arithmetic and logical blocks encapsulates three major decisions:

– Width of *key*: In this work, we explored 32-bit and 64-bit data widths for filtering, arithmetic, and logical operations. There are no limitations regarding custom data width selection; since Vivado HLS supports it. Nevertheless, using larger data widths means utilizing more LUT resources. This becomes specifically critical for low-cost FPGAs because they include LUTs with fewer inputs. Overprovisioning data widths can result in area utilization problems and decreased computational power by failing to meet the timing constraints.

– Number of parallel units: The number of units determine how much data-parallelism can be supported. For this purpose, the approach we followed is to determine the data widths according to the width of DDR-3 RAM line. Thus multiple blocks can process a memory line that is composed of multiple elements of data. In this work, one line of DDR-3 RAM is 512-bits and data widths could be either 32 or 64-bits. Thus, 512/32 =16 or 512/64 = 8 units are instantiated in parallel. It is worth noting that in the TPC-H benchmarks that we looked into, we havent hit to the cases where the required data width is more than 512 bits. Since the 512-bit data width is a property of the DDR-3 interface itself, for more data width, the requirements would be to either (i) use a newer/different technology (High Bandwidth Memory(HBM), 3D stacking, hybrid memory cube, etc.) that supports a wider memory interface, or to (ii) lay the data out in parallel DDR-3 channels. In either case, the AxleDB architecture does not have any inherent limitations regarding the bandwidth to memory.

– Pipelining: We designed all supported filtering, arithmetic and logical operations of AxleDB to be fully pipelined, with an initiation interval of 1 cycle. Thus, all query plans that allow pipelining can be fully supported by our filtering, arithmetic logical and aggregation blocks.

For a given filtering, arithmetic or logical operation, each input data can either be compared with another input data from another table, or it can be compared with a constant. For this purpose, scratchpad registers (SPR) are utilized. These registers hold values and allow the aforementioned operators to be applied to the input data and the SPRs. Our filter unit is capable of performing numerous logical operations, including the BETWEEN operator. As an example, in Figure 5(a), the input data array

[5]The DIV operator (for floating point) can be configured to be included in the AxleDB architecture, it is quick to implement it using Vivado HLS, but it requires a large area and latency. Since we only needed to use this operator only a single time in Q14, we decided not the include the hardware for this operator, and to perform this operation in software (on the host) instead.

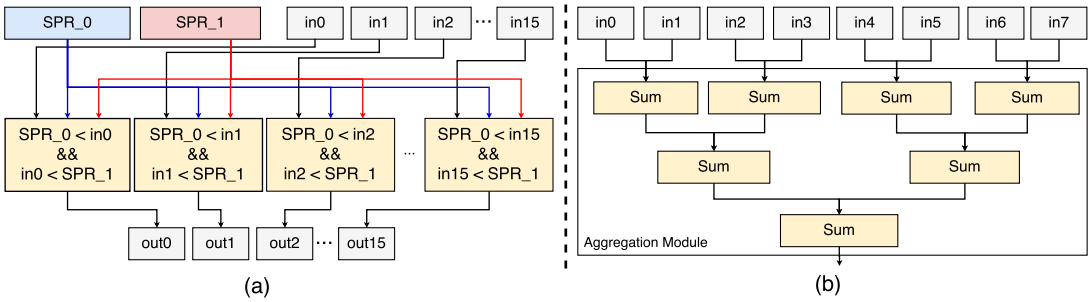


Fig. 5: (a) Filtering blocks that apply the BETWEEN operation to input data using scratch-pad registers (b) SUM aggregation using binary fan-in technique.

holds the column values that require filtering. SPR_0 and SPR_1 hold the filtering qualifiers, the values that input data must be compared against. Thus, input data and SPRs are forwarded to the filters and output are generated. For instance, to perform date *1996-01-11* $<$ *l_shipdate* $<$ date *1995-01-11* BETWEEN operation, SPR_0 holds date *1995-01-10* and SPR_1 holds date *1996-01-12* in 32-bit POSIX time format. For other operations, SPRs work the same way.

4.2. Aggregation Unit

Aggregation operations reduce an input set to a single value. In AxleDB, we provide an n-way aggregator engine that supports MAX, MIN, COUNT, SUM and AVERAGE. Aggregation units are designed with the same three design decisions in mind, which were explained in Section 4.1. They are also fully pipelined with an initiation interval of 1 clock cycle. Similar to the filtering unit, aggregation units are designed to take columns as inputs and to finally combine the results to calculate the final aggregate value. All aggregation operations are implemented in Vivado HLS using the binary fan-in technique. Based on the input array, the size n binary fan-in depth is log2n, and n-1 operators are necessary to form the operator tree. An example is presented in Figure 5(b), where for 8 input elements, 7 sum operators are instantiated to generate a single pipelined result.

Multiple filtering/aggregation blocks can be exploited in two ways: pipelining or time-multiplexing. In a pipelined design, streaming data through multiple instances of the accelerators achieve multiple operations in a single pass. However, for area efficient designs, a single accelerator can be used in a time-multiplexed way. For the studied benchmarks, instantiating multiple filter/aggregation blocks in a pipeline provides the maximum throughput, as we further detail in Section 6.1.

4.3. Hash-Based Units: Table Join and GroupBy

For table joins and groupby operations, we proposed an efficient hashing-based engine. As a first step for table joins, a hash table is constructed using a hash function over the *key* (Build phase). Later on, once constructed, the entries of the second table are probed against this hash table to generate the resulting joint table (Probe phase). For the groupby operation, building a hash table over the *key* can already result in the desired output data.

Hash collisions are critical in hash-based table join and groupby operations because they can be detrimental to good performance. Hash collisions indicate a situation where different keys refer to the same index of the hash table. To resolve collisions, software fallback [6] is a promising solution, but it causes extra overheads. Alternatively, collisions can be managed in hardware as well, by chaining the colliding hash

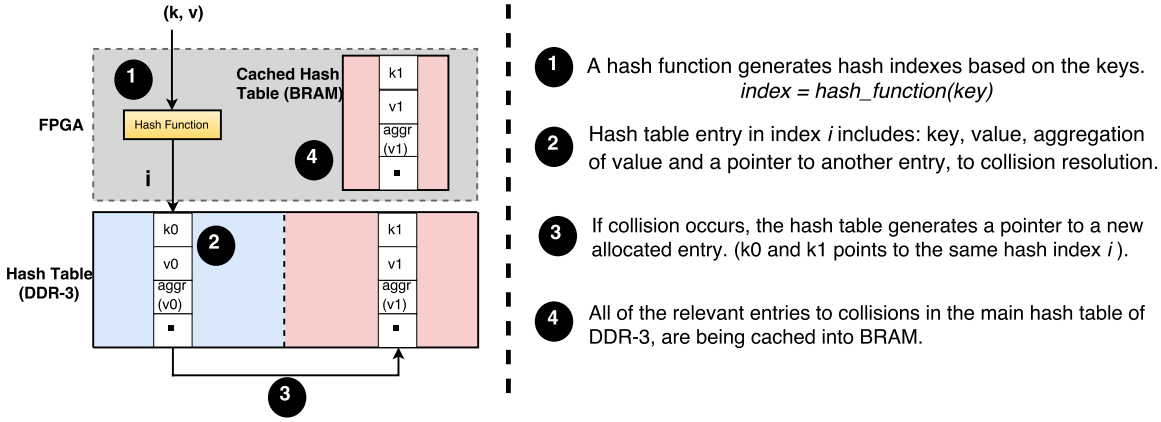


Fig. 6: Hash table caching for enhanced acceleration of table join and groupby. The DDR-3 that locates the hash table is partitioned into two parts. The first part is indexed and uncached and the second part is only used for collisions and is cached inside the FPGA. (k_0 and k_1 are the colliding keys).

table entries [20]. Such behavior can cause pointer chasing and undermine good performance, especially if the hash table is located in high latency DDR-3 memory. To alleviate this problem, thanks to the architecture of FPGAs, we proposed a pure hardware solution. In contrast, the previous techniques such as [6], whereas only off-chip memory is exploited, we targeted to cover the high latency of DDR-3 RAM by exploiting the low-latency Block RAM resources of the FPGA. However, since the size of BRAMs is not large enough to cope with entire hash tables, we present a hash table caching mechanism. We presented an early version of the hash table caching technique in [21]. The cache in BRAM includes some of the hash tables entries that were recently used. Faster access to the cached entries provides a more streamlined execution. Consequently, utilizing DDR-3 RAM and Block RAMs enables us to both, ***(i)*** to use the full capacity of the DDR-3 RAM to store entire large hash tables, ***(ii)*** and to exploit Block RAMs as a hash cache.

As it can be seen in Figure 6, the main components of the proposed engine area Linear Feedback Shift Register(LFSR)-based hash function that allows constant pipelining, Block-RAMs of FPGA as the cache, and the logic of the table join/groupby mechanism. We can store values (for table join), or an aggregation of values (for groupby) inside the hash table, in conjunction with the *key* and a pointer to another hash tables entry for collision resolution. This way, exploiting Block RAM resources, the cache is used to avoid slow pointer chasing effectively. It contains direct-mapped copies of hash tables entries that are recently accessed after a collision. We used the Least Significant Bit (LSB) of the hash index as the index to the cache entry, and the Most Significant Bit (MSB) as a cache tag to discard false positives of cache accesses. The proposed mechanism could also work in conjunction with the multi-threaded hash join engines [22]. The performance of our caching technique is evaluated in Section 6.1.2.

For any hash collision, in the worst case, we need to perform an additional memory read to access the corresponding hash table entry in the chain, which incur an 8-16 clock cycles latency. This is the average latency of the memory part in our platform. However, in the best case, when the read operation of the hash table entry is hit in the cache, (in BRAM) only one cycle is enough to access it.

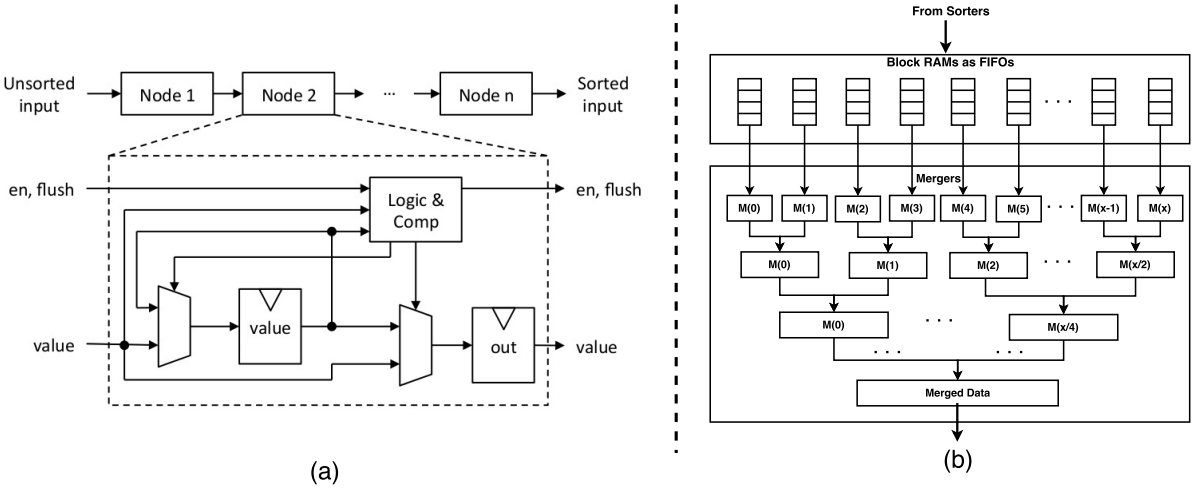


Fig. 7: The architecture of sort engine in AxleDB. (a) Spatial sorter. (b) merge-sort tree.

By the way, the hardware-software co-designed bloom-filter-based hash join accelerator is proposed in [23] that can be a complementary solution for the introduced hash table caching technique in this paper. We do not use several hash functions, and we do not run into false positive problems as with the Bloom filter approach. In contrast, we utilize almost most of the free BRAMs of the FPGA as a cache to achieve an improved throughput. In [23], as Bloom filters use BRAMs to store the bit vector, the processing of large data tables may be limited by the size of BRAMs.

4.4. Sorting and Merging Unit

To efficiently sort large datasets, we used two hardware structures in AxleDB. ***(i)*** The sorter is an extension of the spatial sorter [24], which allows AxleDB to effectively support the LIMIT operation. Furthermore, ***(ii)*** we implemented a merge-sort tree to merge partially sorted sets to be able to sort large input sets.

4.4.1. Spatial Sorter. A spatial sorter is composed of a chained array of sorting registers, each of which effectively performs a compare and swap operation. As it can be seen in Figure 7(a), each sorter node is made up of a comparator, two registers, and two multiplexers [25]. New elements are inserted at the beginning of the sorted array. At each clock cycle and on each node, an input value is compared with the current value. The larger value is stored in the sorter node, and the smaller value is passed on to the next node. An n-node spatial sorter has a time cost of 2n cycles to sort an input set of size n. For this, for n cycles, n tuples are pushed in one by one at the head of the sorter. Later on, in cycle n, a flush signal is pushed in at the head node. Until cycle 2n, this signal propagates until the end of the sorting pipeline. Meanwhile, the tuples that last entered are shifted deeper in the pipeline until they find their correct final positions. By 2n cycles, all the tuples that reside in the nodes of the pipeline are sorted. Then it is possible to start pushing out and writing back the resulting set, and at the same time, to begin receiving a new unsorted input set. The resource usage of the spatial sorter is proportional to its number of inputs. For our case where we want to support wide data inputs, the spatial sorter is a resource-friendly solution.

4.4.2. Merging Sorted Sets. To merge-sort input sets that are larger than the sorter size, the sorter was coupled with a merging tree, as shown in Figure 7(b). N-way input

buffers feed the merger with sorted sets. At each cycle, the smaller input data passes through on all nodes, therefore in the lowest node, the smallest data is output. Block RAMs of FPGA were used as input buffers to sustain constant throughput under DDR-3 RAM latency.

4.4.3. Sorting with the LIMIT Operation. An inherent advantage of employing a spatial sorter is the LIMIT operation. The LIMIT operation is very commonly used in database analytics. We profit from this fact to have a sorter that can do the LIMIT operation at O(n) complexity. However, the best property of this kind of sorting hardware is that it is very regular, chaining-friendly, and maps very well on the FPGA fabric. The LIMIT operation returns only the top n elements of a sorted set. The spatial sorter of size n, when constantly fed with input tuples each cycle (regardless of its size), essentially acts as a LIMIT n operation.

After passing the whole input set through the sorter pipeline, the flush signal will return the sorted top n elements of the set. Therefore, on AxleDB, sorting with a LIMIT operation (where $n <$ sorter_nodes) has a linear time cost of n cycles and furthermore does not require a merging operation afterward. We need to ensure that in a LIMIT n operation, the n results fit inside the sorter nodes, and do not overflow. In the case of overflow, a complete sort (that also uses the merge tree) would be needed. Later in Section 6.1.3, we demonstrate this operation, working with query Q03.

4.5. Block-Level MinMax DataBase Indexing Unit

Database indexing is a technique that improves the speed of retrieving the database tables and is widely used in software DBMS [26], [27], [28]. Indexes are meta-data of the original dataset that are used to locate data quickly. Although there is no standard for generating database indexes, the most commonly used types are B-Tree, Bitmap, and MinMax [28]. In this paper, we focused on the MinMax indexing technique, because it has a straightforward and efficient idea behind, and in addition, it does not suffer from the usual overheads of the other indexing methods, i.e., the large size of indexes in B-Tree method[6], or the low utilization of Bitmap indexing for high-cardinality data method. Some of the software DBMS are already equipped with this technique in different naming: BRIN (Block Range Indexes) in PostgreSQL [29] or Storage Indexes in Oracle [30]. On the other hand, although database indexing is supported in one of IBMs FPGA-coupled products, Netezza [31] which is called zone maps, to the best of our knowledge, it is not investigated thoroughly in FPGA research community yet.

MinMax indexing is an access method intended for the fast scanning of data tables in SSD, by avoiding accesses to the unneeded blocks. We propose a block-level version that uses min and max (b_{Min} ,b_{Max}) values for each data block. As a pre-step of query processing, the aim is to determine data blocks that according to their index values, will or will not pass the filter. This way, the aim is to avoid retrieving those unnecessary blocks from SSD.

In AxleDB, in database creation time, the index values are generated and stored on the SSD. Thus, SSD is partitioned into two distinct parts: data blocks including data tables, and index blocks including index values. The supported filtering operations are LESS THAN ($<$), MORE THAN ($>$) and BETWEEN ($<>$). The functionality of the index checking unit is shown in Listing 1, for a single block i. Depending on the filtering operator, only and if only these conditions are satisfied, it would be necessary

[6]In Section 6.2, we reported some experimental results from PostgreSQL equipped with B-Tree indexing. We observed that in some queries, the I/O transmission of indexes dominates the original dataset execution, causing significant performance degradation.

to retrieve the block i from SSD. (bi_{Min} and bi_{Max} are the min, max index values of data block i. q_{Min} and q_{Max} are the filtering qualifiers.)

Listing 1: The functionality of index checking function for a sample block i.

OPERATION	Index Checking Function
LESS THAN(Field<q_{Max})	(q_{Max}>=bi_{Min})
MORE THAN(Field>q_{Min})	(q_{Min}<=bi_{Max})
BETWEEN(q_{Max}<Field<q_{Min})	(q_{Max}>=bi_{Min})&&(q_{Min}<=bi_{Max})

The index checking function is repeated for all of the indexes, thus, finally we obtain the list of only the necessary blocks of data. Although the total size of indexes is proportional to the number of data blocks (size of total data divided by block size), as we only store two values for each block (b_{Min},b_{Max}), the size of index storage is mainly negligible, leading to a small performance overhead. We exploited local Block RAMs of the FPGA to store indexes. The following steps summarize running AxleDB with MinMax indexing capability, as a pre-step of query processing:

(1) AxleDB initializes the index checking unit with one of the supported operations ($<$, $>$, $<>$) and their qualifiers (q_{Min},q_{Max}).
(2) AxleDB first retrieves the index blocks from SSD. Later on, for each particular index that corresponds to a particular data block, AxleDB checks it (as shown in Listing 1). Accordingly, the list of the required data blocks for the further processing are generated in the Block RAMs.
(3) Once the index checking process is completed, AxleDB continues the query processing by retrieving only those necessary data blocks that are listed in Block RAMs.

The efficiency of MinMax indexing technique highly depends on the data distribution model. For instance using this technique, for the highly ordered dataset, most of the unnecessary data blocks can be detected, which in turn leads to a significant reduction in I/O transmission. We further analyze it in Section 6.1.4.

5. EVALUATION METHODOLOGY

5.1. Configuration of the AxleDB

We developed AxleDB on a VC709 FPGA development board with an XC7VX690T FPGA and 4GB of DDR-3 RAM. It accesses a Crucial M4-256GB SSD through a customized version of an SATA-3 controller, based on Groundhog [12]. We used a relatively large block size (512 KB), which can help minimize the SSD overheads and, thus lead to significant improvements in data transfer throughput. AxleDB is directly attached to the host through a high-speed PCIe-3 interface for data/instruction transmission. Our accelerators and DDR-3 RAM controllers run at 200 MHz, PCIe-3 controller at 250 MHz, and SATA-3 at 150 MHz, therefore we used various synchronizing FIFOs for clock domain crossing.

In order to thoroughly evaluate the efficiency of the various components of AxleDB, we ran the benchmarks in two modes: ***(i)*** cold run, where the input datasets are originally located inside the SSD, and ***(ii)*** warm run, without considering the I/O time of SSD, and assuming that the datasets are already loaded in the DDR-3 memory of the platform. Thus, in the warm mode, the total processing time of the queries does not include the time of loading input data tables from SSD to the DDR-3. To better analyze the cold and warm runs, we partitioned the total execution time of the query into three parts: ***(i)*** the I/O time of SSD, i.e. the required time of transferring input datasets from the SSD to DDR-3 memory of AxleDB, ***(ii)*** execution time, i.e. the query execution time of the processing units (accelerators) of AxleDB, and ***(iii)*** the time spent on the other

parts, including query planning time[7], the time for PCIe-3 data transfers, and finally the overhead of device controllers. The last portion is negligible for large scales of data.

5.2. Configuration of Comparison Cases: MonetDB, PostgreSQL and CStore

We evaluated AxleDB against the query processing engines of several state-of-the-art software DBMS: ***(i)*** MonetDB 11.21 [9] as a popular column-oriented database system, ***(ii)*** PostgreSQL 9.5 (PGSQL) as a popular object-relational row-oriented database system [8], and ***(iii)*** CStore as the PostgreSQL's column-oriented data store extension [10]. More specifically, it is worth noting that:

— **MonetDB** has several unique features to optimize the I/O and computation, simultaneously: ***(i)*** it is built on a column representation of database relations, ***(ii)*** it has an innovative storage model based on vertical fragmentation, ***(iii)*** it has a CPU-tuned query processing architecture, ***(iv)*** it exclusively tries to use the main memory for the processing, and ***(v)*** it has the capability of running queries in a multi-threaded fashion.
— **PostgreSQL** is intrinsically a row-oriented database system. It is equipped with a wide set of database indexing methods, such as BRIN, B-Tree, etc., that can be used as an appropriate comparison case with the proposed FPGA-based MinMax indexing technique in AxleDB. Also, to get better performance, PostgreSQL is extended with an extra patch to support fixed-decimal data type [33]. Fixed decimal is a fixed precision decimal type which provides a subset of the features of PostgreSQL's built-in NUMERIC type, but with increased performance.
— **CStore** is an extension that enables column-oriented data storage in PostgreSQL. It uses the Optimized Row Columnar (ORC) format, which brings some benefits such as; compression, column-projection, and skips indexes (similar to MinMax/BRIN).

We run MonetDB, PostgreSQL, and CStore on a Supermicro server machine, equipped with two E2630 Intel Xeon processors, with a total of 12 cores and 24 threads, running at a maximum frequency of 2.3 GHz. The server is attached to a Crucial M4 SSD disk (as in the AxleDB) to store database tables. The operating system (OS) is Ubuntu 64-bit 12.04, with kernel version 3.13. To have a fair comparison, we equip the server with the same size of the memory with AxleDB, 4GB DDR-3. However, we observe a system crashes of the PostgreSQL/CStore runs for the 10GB scale benchmarks, which is the consequence of the insufficient system memory. Thus, for this special case, the server is equipped with a larger capacity of memory. We observed that at least 16 GB is enough to accomplish the query processing without system crashing. Consequently, in summary, the host is equipped with 4GB and 16GB for MonetDB and PostgreSQL/CStore experiments, respectively. Furthermore, similar to the AxleDB, software DBMS experiments were also ran in two modes, ***i)*** cold mode, where input data tables are located in the SSD, and ***ii)*** warm mode, where input data tables are already loaded into the DDR-3 memory from the original database storage, SSD. To obtain their execution times, we ran each query twice, consecutively. The first run is in the cold mode. In contrast, the second run is executed using internal buffers, where the data is already located inside the main memory of the server. The second run is in the warm mode.

[7]As the query planning of AxleDB is currently a manual process, thus to have a fair comparison with software DBMS, we extracted the average time of the query planner of MonetDB (8ms for cold and 2ms for warm runs), and used it in this part [32].

5.3. Introducing the Benchmarks Methodology

We evaluated AxleDB with five decision-support TPC-H queries, under various conditions [13]. The studied queries are Q01, Q03, Q04, Q06, and Q14, which heavily utilize and stress the various hardware accelerators. For instance, Q01 requires several aggregation accelerators, Q03 employs nested hash join and sorter operations, and Q06 heavily utilizes the filtering accelerator. Furthermore, the selected queries represent process-intensive (Q01), I/O-intensive (Q06, Q14) and I/O-process-balanced (Q03, Q04) workloads, which allows us to exhaustively analyze the cold and warm runs of the platforms. This classification is based on running the TPC-H queries in default PostgreSQL, equipped with B-Tree indexing, and on a host machine with large enough memory to prevent memory thrashing. We elaborated them in Section 6.2.

The TPC-H database generator allows generating the input dataset in various scales. Using this capability, we analyzed the AxleDB in 1GB and 10GB scales, to evaluate it while dealing with small and large datasets. Regarding the maximum size of the data that can be handled in AxleDB, there are two constraints:

— **The size of the input data tables:** The maximum size of the input data tables that AxleDB can handle is constrained by the size of the data storage (DDR-3 for in-memory- warm runs- case and SSD in other cases- cold runs).
— **The size of the hash table:** Other constraint is the size of the hash table that needs to be fitted the DDR-3 size. This is because our hash join/group-by module uses DDR-3 as the hash table and it does not support yet the extremely large hash tables that their size exceeds the DDR-3 size.

It is worth noting that for the studied queries in the experimental results in Section 6, the aforementioned limitations were never observed. However, attaching larger DDR-3 RAM to AxleDB, it can cope with larger scales. In addition, other promising solutions include ***(i)*** supporting multiple disks through the use of daughter-cards on the FPGA board [34], ***(ii)*** applying range partitioning on the database tables [35], or ***(iii)*** having a distributed framework [36].

5.4. Introducing the Evaluation Metrics

For each given query, we compare AxleDB against the software baselines in terms of speedup, throughput, and energy dissipation metrics. The speedup shows the relative query processing time of the query, throughput in an absolute metric and shows the amount of the processed data in one second, and the energy is used to compare the relative energy dissipation (power * time) of each query. These metrics are defined in equations 1, 2, 3, and 4:

$$speedup = \frac{query\_processing\_time\_of\_sw\_platforms(sec)}{query\_processing\_time\_of\_AxleDB(sec)} \tag{1}$$

$$power\_efficiency = \frac{power\_consumption\_of\_sw\_platforms(watts)}{power\_consumption\_of\_AxleDB(watts)} \tag{2}$$

$$energy\_efficiency = speedup * power\_efficiency \tag{3}$$

$$throughput(absolute) = \frac{total\_amount\_of\_processed\_data(MB)}{query\_processing\_time(sec)} \tag{4}$$

6. EXPERIMENTAL RESULTS

In this section, first, we individually evaluated the components of AxleDB. Later on, we presented the experimental results of the queries under test and discussed on their

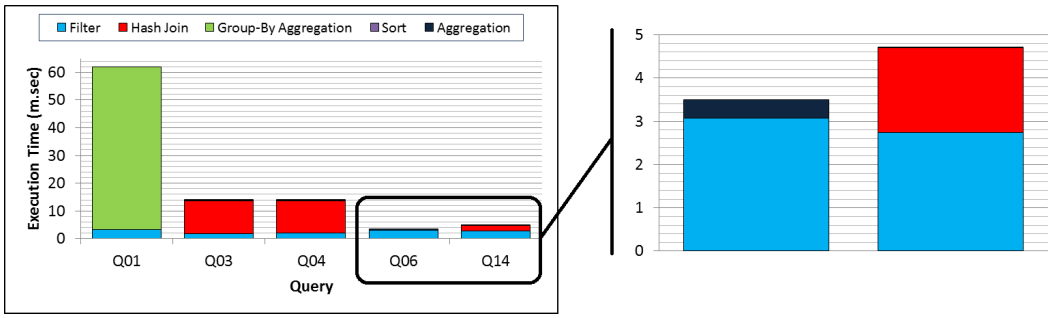


Fig. 8: The total execution time of the queries, partitioned into per-accelerator.

performance, including comparisons with multi-threaded DBMS. Finally, we reported and discussed on the power and energy consumption of AxleDB. Later on, the utilization rate of hardware resources is discussed.

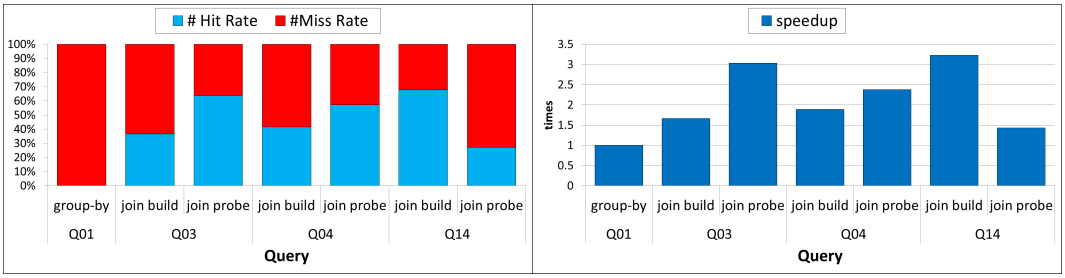
6.1. Evaluating Query Accelerators of AxleDB

We individually evaluated the query accelerators of AxleDB, including filter, aggregation, hash join/groupby, sort, and MinMax indexing. For each of the given queries, we partitioned the total execution time into the time spent in each of the accelerators, as reported in Figure 8. It is worth noting that, only the query execution time is included in this figure, without considering the I/O time of SSD, PCIe-3, or the overhead of controllers. We will evaluate those parts in the Section 6.2.

6.1.1. Evaluating Filtering and Aggregation Accelerators. In Sections 4.1 and 4.2, we presented the design idea behind the filtering, arithmetic, logical, and aggregation units. Before using these units in the AxleDB, we first verified the functionality and also the I/O interfaces, in Vivado HLS [18], [19]. To enable full pipelining, all arrays are completely partitioned using Vivado HLS. To improve performance, the execution latency of all operations take one clock cycle, except for multiplication which takes six clock cycles. For the queries under test, the filtering of char arrays is handled by the filter blocks treating the strings as aggregate 8-bit char arrays. However, for regular expression types of filtering operations, more advanced hardware would be required, which could be incorporated in AxleDB if desired. As it can be seen in Figure 8, among the studied queries, the filtering operation is dominant in Q06 and has a significant contribution in Q14, as well. In contrast, for the other queries, as the filtering operation passes a large portion of data to the further expensive operations, its contribution in total execution time is relatively reduced.

6.1.2. Evaluating Hash Join and Groupby Accelerators. We evaluated the hash-based join and groupby on the given TPC-H queries, except for Q06 that requires neither of them. In our hardware platform, DDR-3 RAM is large enough to store all the hash tables used in this study. Our Block RAM-based cache size is 2.8 MB, which utilizes more than half of the available Block RAMs of our FPGA device.

We compared our results against a baseline that does not employ the cache, and all accesses are served from DDR-3 RAM in a pipelined fashion. Figures 9a and 9b show the experimental results for the given queries in 10GB scale, in terms of the hit/miss rate of the cache and the speedup of hash join/groupby equipped by the hash table caching mechanism in comparison to the cache-less baseline version, respectively. On the experiments of 1GB scale, we observed that the cache is not utilized as there are no hash collisions and also hash table fits in the Block RAMs. For those queries



(a) (b)

Fig. 9: Evaluation results of the proposed hash join engine. (a) Hash table caching results: cache hit ratio. (b) Hash table caching results: the speedup comparison against the baseline without any hash table caching.

with several join/groupby operations, such as Q03, we run all the involved operations separately and later on, their summation is depicted in this figure, as the final report of each particular query. The performance gains employing caching can be perceived as a direct consequence of the cache *H.R* (up to 63.1%). Accordingly, as it can be seen in Figure 9b, that the speedup varies from 1.4X to 3.2X for the given queries. Although in our experiments the improvement limitation is mainly due to the limited cache size, the Block RAM capacity in modern FPGAs is in the order of tens of MBs, which could help build larger caches.

Analyzing the breakdown of query execution time in Figure 8, we can see that hash-based operations dominate the execution time in some of the given queries: i.e. the groupby operation in Q01 and the hash join operations in Q03 and Q04.

6.1.3. Evaluating the Sorting Accelerator. The sorter performance and area usage is a function of how deep the sorter is and how wide the merger circuitry is. Theoretically speaking, memory capacity and bandwidth limitations apart, there is no limit on how many elements could be sorted using such a sort-merge scheme. For n elements to sort, our approach takes 2n cycles using only the sorter module, and an additional n cycles for each n-way merge tree run. Eg. If we have an 8-node sorter and a 4-to-1 merger, to sort n=32 elements, it takes (32x2 (sort) + 32 cycles (merge)) cycles, which is 3n cycles in total, plus latency. But for n=128 elements, it would take 128x2 cycles (for sort), but we still have 128/8=16 partially sorted sets to merge, which can be done with 4 iterations of 4-to-1 merge plus a final iteration (again, of a 4-to-1 merge), which takes a total of 5n cycles: so the whole sorting operation takes 7n cycles in total, plus latency.

More specifically, working with Q03, the LIMIT operator permits us to require a sorter that only needs to be larger than 10 nodes, and no sort-merger circuitry is needed. The spatial sorter needs to support at least a 16 Bytes *key* and a 16 Bytes *value*. However, Q01 requires support for wider data (16 Bytes *key*, 48 Bytes *value*), as well as a deeper sorter unit. In Section 6.3, we elaborate more on our final circuitry of choice for running our experiments. For the other queries, the sorter module is not needed, as they do not have any ORDER BY operation.

6.1.4. Evaluating MinMax Indexing Technique. To evaluate our FPGA-based indexing method, we run the benchmarks with MinMax indexing in software DBMS, called BRIN (Block Range Indexes) in PostgreSQL and skip indexes in CStore. As illustrated in Section 4.5, the MinMax indexing method can be effectively utilized in the highly-

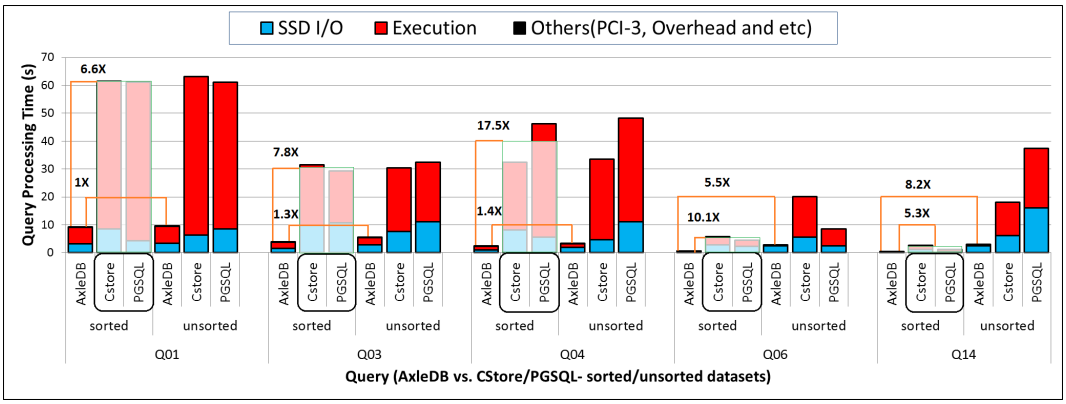


Fig. 10: Performance results for the studied benchmarks: MinMax indexing in AxleDB vs. BRIN indexing in CStore and PostgreSQL, for sorted and unsorted (default) 10GB scale dataset. Lower is better.

ordered datasets. However, TPC-H is a synthetic benchmark with mainly a normal distribution of data. Thus, to investigate the efficiency of the MinMax indexing method, we generated sorted versions of the data tables of the given queries and ran them on AxleDB/CStore/PostgreSQL using the modified tables. These tables are sorted based on those *keys*, which are also used in the filtering operation of the given queries. For instance, Q03 is sorted and also indexed based on *l_shipdate*.

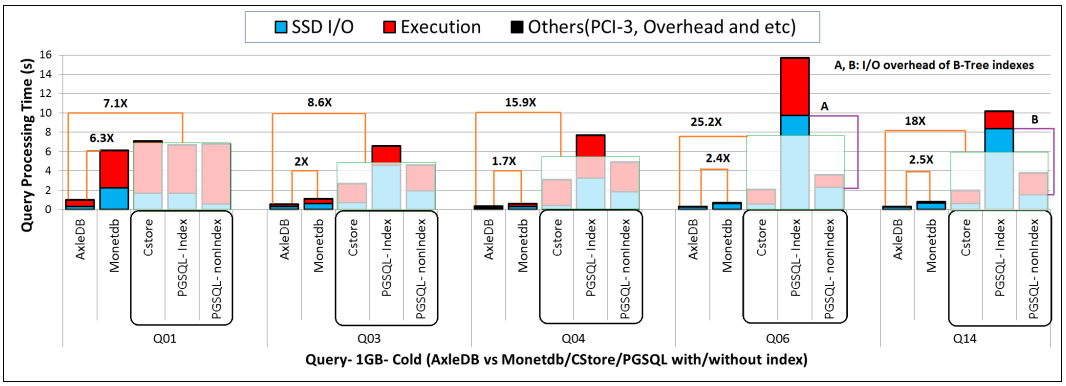
Figure 10 presents the total execution time of the queries in 10GB scale, which is broken down into the aforementioned partitions. In general, we observed that all of the platforms (AxleDB, PostgreSQL, and CStore) could process the given queries on sorted datasets faster than the unsorted versions, thanks to the inherent characteristics of the MinMax indexing method. However, the speedup varies for different queries. More specifically, we observed that:

— Q01 is a process-intensive query. Thus, the indexing method is not efficiently utilized (for instance, we saw that there was no speedup of AxleDB comparing sorted vs. unsorted datasets). However, we observed a 6.6X speedup comparing AxleDB against PostgreSQL and CStore.
— in Q03 and Q04, the total query processing time of AxleDB is reduced proportionally, up to 40%.
— in Q06 and Q14, the total query processing time is significantly reduced in all the platforms (for instance, 5.5X and 8.2X speedup of AxleDB on sorted vs. unsorted datasets). These queries are I/O-intensive, thus, skip loading most of the unnecessary parts of data lead to the considerable speedup, thanks to the efficient utilization of the indexing method.

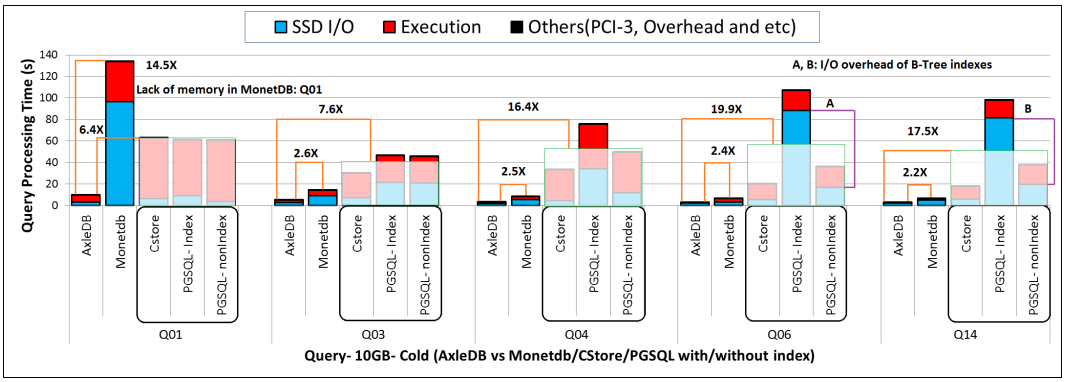
In AxleDB, the utilization of MinMax indexing method not only improves the I/O throughput but also reduces the overall query execution time. For instance, as it can be seen in Figure 10, running AxleDB for sorted vs. unsorted datasets, we observed a 5.8X speedup of the execution time in Q06.

6.2. Overall Performance Analysis

In this section, we compared the query processing time of AxleDB against MonetDB, CStore, and PostgreSQL under various conditions. For the overall performance analysis, we ran the platforms in default mode: ***(i)*** MonetDB without any indexing, ***(ii)***



(a)



(b)

Fig. 11: Total query processing time of the studied benchmarks in cold mode, comparing AxleDB vs. MonetDB, CStore, and PostgreSQL. (a)1GB scale, (b) 10GB scale. Lower is better.

CStore that is equipped with an embedded MinMax indexing (called skip indexing), ***(iii)*** PostgreSQL, which is equipped with B-Tree indexing, as well as an index-free version, and ***(iv)*** AxleDB with the MinMax indexing method. The experimental results of cold and warm runs of the platforms, on 1GB and 10GB scales, are shown in Figures 11 and 12, respectively.

6.2.1. Evaluation of Cold Runs. Figure 11 presents the total processing time, broken down into the aforementioned partitions. On average, AxleDB can process queries more than an order of magnitude faster than CStore and PostgreSQL (15X in 1GB and 13.6X in 10GB scales), as well as showing speedup against MonetDB (3X in 1GB and 4.7X in 10GB scales). More specifically, among the set of studied queries, we observed that for:

— **process-intensive queries** (Q01), as it can be seen in Figure 11(a), for 1GB scale, the I/O time of SSD is negligible, the indexing method is not utilized well, and the execution time is dominating. In the process-intensive workloads, the performance gain of the AxleDB is mainly the consequence of exploiting highly efficient query accelerators, in a deeply pipelined fashion. For this particular query, the speedup is 6.3X compared to MonetDB, and 7.1X against the different versions of PostgreSQL, including CStore, PostgreSQL-indexed, and PostgreSQL-non-indexed. Although for

10GB scale, as shown in Figure 11(b), AxleDB, CStore and PostgreSQL expose similar behaviors, we observed that MonetDB is not optimized very well, as it frequently accesses the SSD to retrieve data. This is the result of insufficient memory to store temporary data for this particular memory-dependant process-intensive query.

— **I/O-process-balanced queries** (Q03, Q04), the improvement of AxleDB is the consequence of both I/O efficiency and faster execution. For instance, AxleDB reduces SSD I/O time by 17.9X and execution time by 31.5X for Q04 in 1GB scale, comparing to the index-enabled PostgreSQL, which leads to a total of 23.4X speedup for this particular case. For these queries, on average, the speedup is 1.8X for 1GB and 2.5X for 10GB scale, against MonetDB, and 12.2X for 1GB and 12X for 10GB scale, against different versions of PostgreSQL.

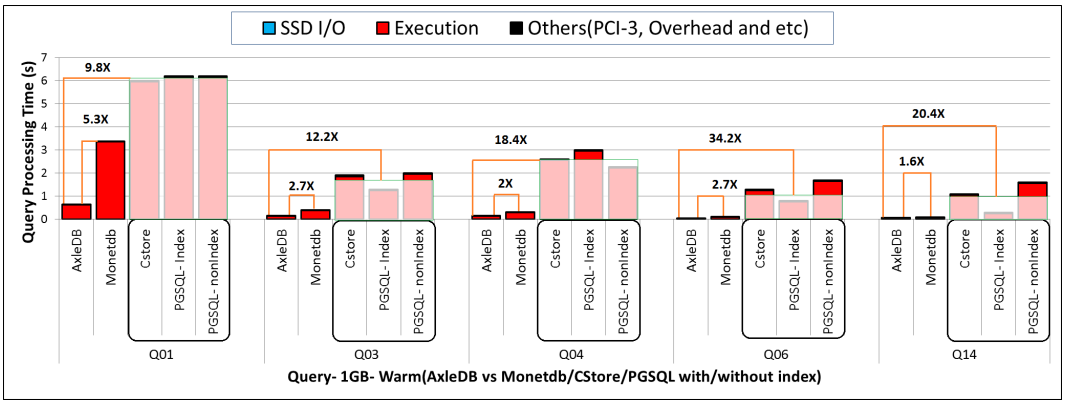
— **I/O-intensive benchmarks** (Q06, Q14), SSD I/O time is dominating. Comparing PostgreSQL with index-enabled vs. non-index versions, we unexpectedly observed a significant overhead of B-Tree indexes, which causes substantial performance degradation. In contrast, AxleDB, CStore, and MonetDB, thanks to their column-oriented data storage, significantly reduce SSD I/O transfers. The results demonstrate that AxleDB can process these queries, on average 2.4X and 2.3X faster than MonetDB, and 21.6X and 18.7X faster than the average of different versions of PostgreSQL, for 1GB and 10GB scales, respectively.

In summary, the results clearly demonstrate that AxleDB achieves the acceleration of query processing, thanks to the pipeline-optimized query accelerators, and simultaneously optimizes the SSD I/O performance, thanks to the data movement techniques that were used, such as direct attached, column-oriented data storage and database indexing.

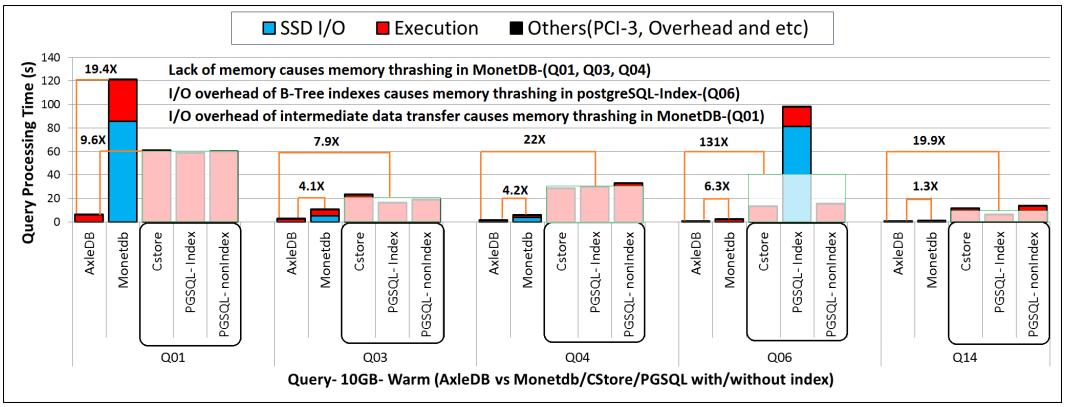
6.2.2. Evaluation of Warm Runs. For the warm run mode, the speedup of the AxleDB compared to the software platforms is the consequence of the pipelined execution of the query accelerators in the AxleDB. Furthermore, as it can be seen in Figure 12(b), in 10GB scale and for some of the queries, the lack of memory in the host causes memory thrashing that leads to a performance degradation of software platforms. More specifically, for different scales of datasets:

— In 1GB scale, as it can be seen in Figure 12(a), we observed that the attached memory of the platforms is large enough to already store the small-sized data. Thus, the SSD I/O is totally skipped. Consequently, the speedup of AxleDB against software platforms is the sole result of a faster query execution in AxleDB. The speedup of AxleDB is from 1.6X to 5.3X against MonetDB (on average 2.9X), and from 9.8X to 34.2X (on average 19X) against the different variants of PostgreSQL.

— In 10GB scale, as it can be seen in Figure 12(b), we observed several exceptions, where the memory capacity of the platforms is not sufficient to store entire datasets. ***(i)*** For MonetDB, the SSD I/O time contributes to 71%, 52% and 64% of the total query processing time for the queries Q01, Q03, and Q04, respectively. This I/O overhead is the result of the extra data (parts of the input dataset or temporary data that is needed during the processing) retrieved from SSD. ***(ii)*** On the other hand, for PostgreSQL, we observed memory thrashing for the index-enabled version of Q06. This overhead is the result of a large amount of memory used for the indexes. For this particular case, the I/O contributes to 82% of the total query processing time, which, in turn, executes significantly slower than AxleDB (304X). Eventually, the speedup of AxleDB ranges from 1.3X to 19.4X against MonetDB (on average 7.1X), and from 7.9X to 131X (on average 38.1X) against the different variants of PostgreSQL.



(a)



(b)

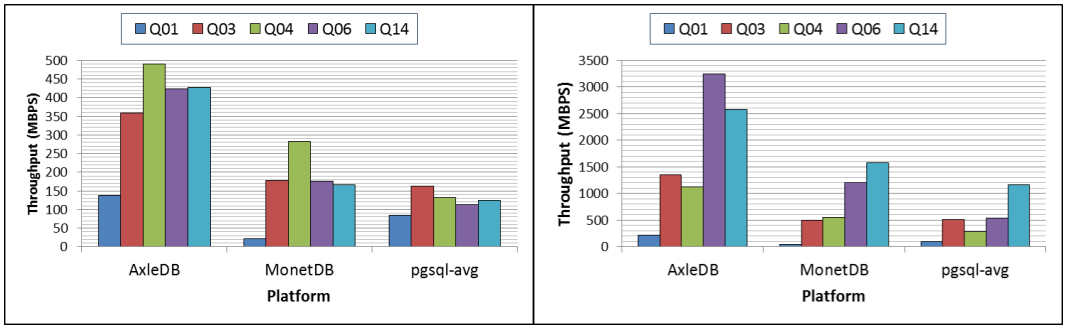
Fig. 12: Total query processing time of the studied benchmarks in warm mode, comparing AxleDB vs. MonetDB, CStore, and PostgreSQL. (a)1GB scale, (b) 10GB scale. Lower is better.

In this section, to evaluate the cold and warm runs, we used a single-threaded version of software platforms and observed a significant improvement using AxleDB. For further investigations, we will analyze their multi-threaded version in Section 6.4.

6.3. A Discussion on the Optimized points of AxleDB in terms of Data Management and Execution Accelerating

In this section, we analyze the speedups of the warm against cold runs of AxleDB against the comparison cases for the studied queries. In general, the significant speedup of AxleDB against software-based comparison cases is the consequence of two optimization points: ***i)*** offloading the query processing onto the FPGA and following the streamline dataflow execution model and ***ii)*** Optimized accesses to the SSD (tightly coupled to the processing units -accelerators- in the FPGA). Accordingly, we analyze their impact in the speedup of AxleDB for each query individually. Toward this goal, by comparing the experimental results in Figure 11(a) (cold runs) and Figure 12(a) (warm runs) in 1GB scale, we observe that for:

— **Offloading the query processing onto the FPGA:** To evaluate the impact of this optimization point, we use the experimental results in Figure 12(a) for the warm



(a) (b)

Fig. 13: Comparing the practical throughput of AxleDB against MonetDB and PostgreSQL. (a) cold runs in 1GB scale. (b) warm runs in 1GB scale. Higher is better.

runs. We observe on average 2.9X and 18.9X speedup of AxleDB against MonetDB and PostgreSQL, respectively, in 1GB scale. Since there is no memory thrashing in the 1GB scale of warm runs, we can conclude that these speedups are purely the consequence of the FPGA offloading in AxleDB. Moreover, as it can be seen, MonetDB is more optimized than PostgreSQL especially in Q06 and Q14, which can be the consequence of better memory management (and higher bandwidth utilization) in MonetDB.

— **Performing optimized access to the SSD:** To evaluate the impact of this optimization point, we use the experimental results in Figure 11(a) for the cold runs. As it can be seen in this figure, the total query processing time is partitioned into the SSD I/O and the execution time in the processing units. Comparing only the SSD I/O time (blue part), we observe that AxleDB is on average 3.1X and 10.8X more optimized than MonetDB and PostgreSQL, respectively, in 1GB scale. This can be the consequence of the better SSD I/O management in AxleDB thanks to the tight coupling of SSD to the hardware accelerators. This lets AxleDB skip the loading of the unnecessary parts of the data tables, and provides a higher utilization rate of the SSD bandwidth by using the large block sizes (up to 1MB). Moreover, as it can be seen, the SSD I/O management in MonetDB and CStore version of PostgreSQL is more optimized than the default version of PostgreSQL, as they follow a column-store data format.

6.4. Overall Throughput Analysis of AxleDB against the Software Platforms

In this section, we briefly analyze the throughput of AxleDB, MonetDB, and an average of different versions of PosgtreSQL. The experimental results are shown in Figure 13 (a) and (b) for cold and warm runs, respectively, in 1Gb scale. The throughput is computed as the amount of pure data (input and output data tables) that can are processed in each platform, as formulated in the Equation 4. Another word, we did not include database indexes, intermediate tables e.g. hash tables, etc., to compute the overall throughput in this figure. As it can be seen:

— in the cold runs, the SSD throughput is the bottleneck of the overall throughput that limits it to below 500 MB/s (the maximum theoretical throughput of the SATA-3 interface.). However, we observe better throughput of AxleDB in comparison to the other platforms.

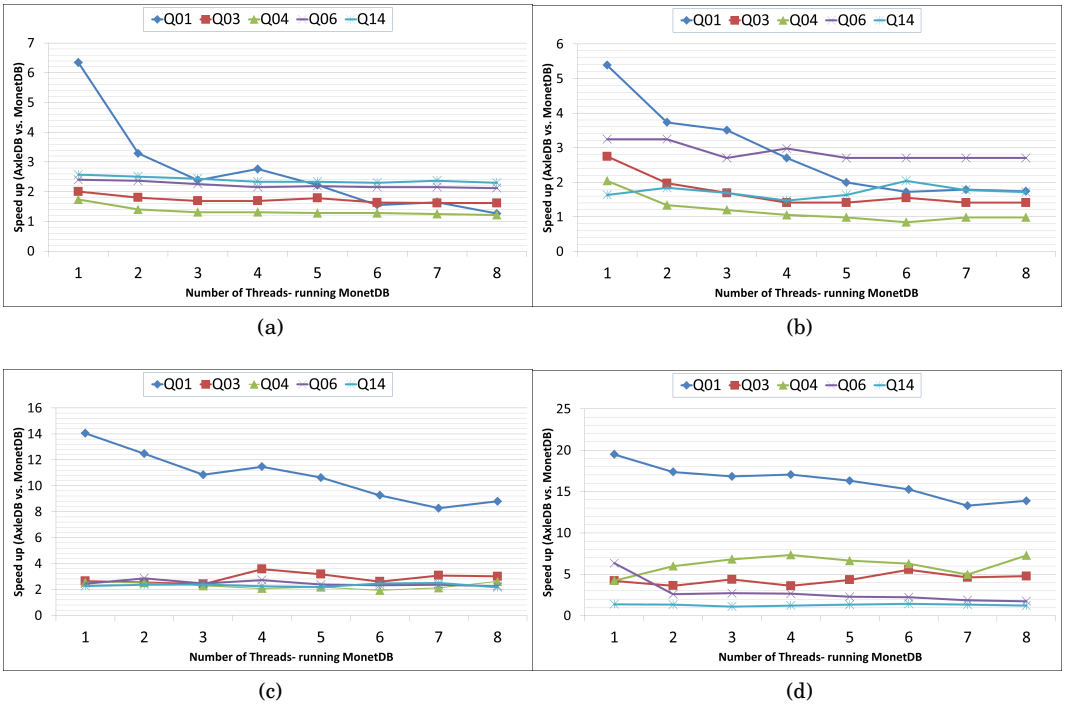


Fig. 14: Comparing the speedup of AxleDB vs. multi-threaded MonetDB. (a) cold runs in 1GB scale. (b) warm runs in 1GB scale. (c) cold runs in 10GB scale. (d) warm runs in 10GB scale. y-axis represents the speedup of AxleDB against the multi-threaded MonetDB- the relative query processing time, as formulated in the Equation 1. Higher is better.

— in the warm runs, as data tables are located in the DDR-3 memory, thus there is no SSD I/O bottleneck, and we observe better throughput up to 3.2 GBPS. However, as it is expected, the intermediate data transmission prevents to achieve the maximum theoretical throughput of the DDR-3 memory. More specifically, we observe this behavior in Q01, whereas as discussed in Section 6.1 and shown in the Figure 9 (a), the groupby aggregation operation dominates the total query processing time, as a consequence of significant intermediate data transmission.

It is worth noting that in AxleDB, MonetDB, and CStore we need to load only the necessary columns of data tables to the FPGA. In contrast, the other row-based versions of PostgreSQL, we load the entire data tables. This design point is already considered in the throughput results.

6.5. Evaluating the Performance of AxleDB against Multi-threaded MonetDB

To analyze the effects of the multi-threading, we compared AxleDB against the multi-threaded version of MonetDB. Unfortunately, as of now, PostgreSQL and its variants do not support multi-threading. Figure 14 shows the experimental results for cold and warm runs of 1GB and 10GB scales, in terms of the normalized speedup of AxleDB vs. MonetDB, exploiting a varying number of CPU threads. We did not observe any

significant changes in the behavior of MonetDB utilizing more than '8' threads. Thus the diagram includes the experimental results up to 8 threads. We observed that:

— For cold runs, as it can be seen in Figure 14(a) and (c), utilizing additional threads does not lead to improving the performance of MonetDB, except for Q01, which is a process-intensive query. Consequently, for Q03, Q04, Q06 and Q14, a constant speedup is achieved, almost independently from the number of threads utilized. In contrast, in Q01, utilizing more threads accomplishes the query execution in a parallel and thus in a rapid fashion, which causes to reduce the speedup of AxleDB vs. MonetDB from 6.3X to 1.2X in 1GB, and from 14.1X to 8.8X in 10GB scales. Furthermore, for 10GB scale as a result of memory thrashing issue, in Q01 the speedup of AxleDB is about an order of magnitude, while for the other queries, it is between 1X and 3X.
— For warm runs, as it can be seen in Figure 14(b) and (d), as there is no SSD I/O transferring time (except those queries where memory thrashing was observed), utilizing more threads leads to better performance of MonetDB. Accordingly, for 10GB scale, the speedup of AxleDB varies between 5.3X and 1.6X, and between 2.8X and 0.98X, against single-threaded and 8-threaded MonetDB, respectively.

In summary, we observed that multi-threading in MonetDB leads to high performance, especially in process-intensive queries. However, as AxleDB simultaneously employs a set of highly-efficient accelerators, as well as optimizing the I/O, it can accomplish the processing of the studied queries faster than multi-threaded MonetDB in most cases.

6.6. Evaluating the Energy-Efficiency of AxleDB against Multi-threaded MonetDB

Table III shows the power consumption of AxleDB, estimated using Vivado Power Estimator after the Place & Route stage. A considerable amount of power is dissipated while interfacing hardware and accelerators. Also, multiple clock domains draw additional power in the data connection switch. On the other hand, to measure the power dissipation of the processor, we used Intels Running Average Power Limit (RAPL) energy meter. RAPL exposes energy usage estimates to software via model-specific registers, using hardware performance counters and I/O models [37].

For the comparison against state-of-the-art DBMS, we considered the total power consumption of AxleDB and software platforms, excluding the power dissipation of data storage devices, i.e. SSD and DDR-3 RAM. We reported the energy efficiency of AxleDB against multi-threaded MonetDB in Figure 15. It is worth noting that the energy consumption (in Joules) is obtained by measuring the power dissipation using RAPL for software platforms and using Vivado tools for AxleDB (in Watts), and multiplying it with the total query processing time (in seconds). Regarding the experimental results, we observed that:

— As it can be seen in Figure 15(a) and (c), for cold runs of MonetDB in 1GB scale, AxleDB is 3.5–14.8X (on average 6.7X), and 2.4–5.3X (on average 3.9X), more energy efficient than the single-threaded and eight-threaded MonetDB, respectively. The improvement for 10GB is more significant, as it varies from 5X to 32.9X (on average 10.8X), and from 4.9X to 23.4X (on average 9.1X). In some cases such as Q01, this is the result of memory thrashing. Furthermore, in cold runs, as the SSD I/O has a significant contribution, the measured power dissipation of computing cores does not considerably vary.
— As it can be seen in Figure 15(b) and (d), for warm runs of MonetDB in 1GB scale, the energy optimization of AxleDB varies from 5.5X to 13.1X (on average 8.1X), and from 2.8X to 10.3X (on average 6.3X), compared against the single-threaded

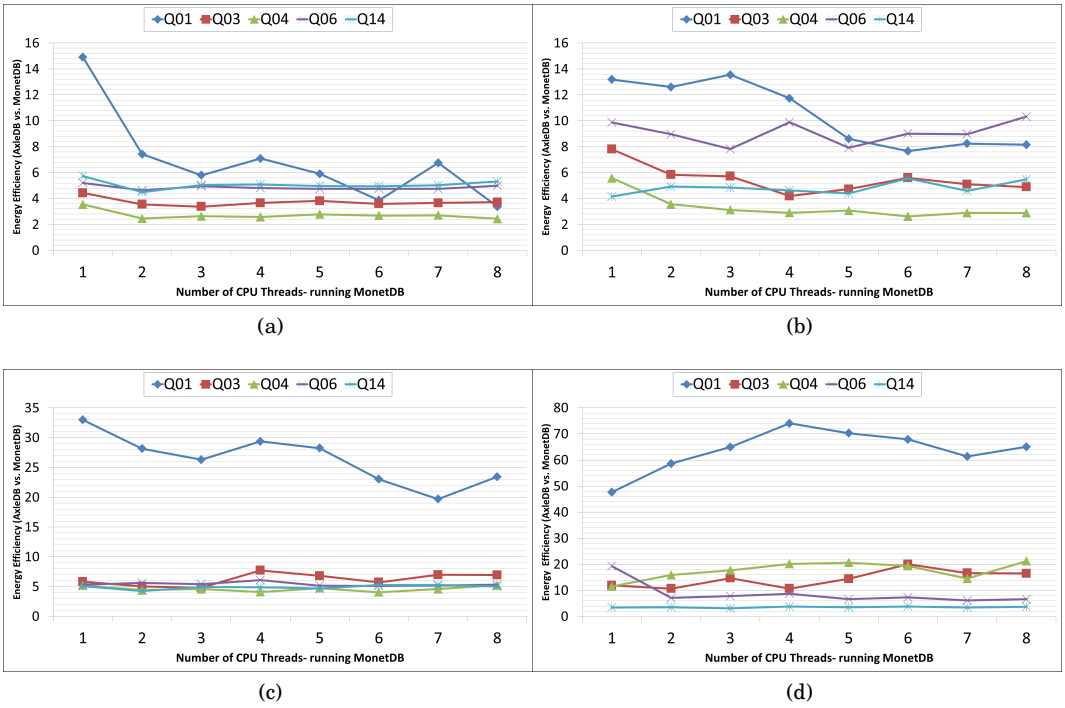


Fig. 15: Comparing the energy efficiency of AxleDB vs. multi-threaded MonetDB. (a) cold runs in 1GB scale. (b) warm runs in 1GB scale. (c) cold runs in 10GB scale. (d) warm runs in 10GB scale. y-axis represents the relative energy efficiency of the AxleDB against the multi-threaded MonetDB- the relative energy efficiency as formulated in the Equation 3. Higher is better.

and eight-threaded MonetDB, respectively. Scaling up to 10GB scale, similar to cold runs, we observed better optimization. Exploiting more threads in warm runs of MonetDB leads to additional power consumption, as more processor cores are active.

Furthermore, compared to the different variants of PostgreSQL, including indexed, non-indexed and CStore, we observed that AxleDB is at least an order of magnitude more energy efficient, on average 25.7X for cold runs and 62.1X for warm runs.

In summary, as AxleDB is inherently faster than software DBMS, thanks to its deeply-pipelined architecture, and is also more power-efficient, thanks to a lower operating frequency than software platforms (on average 200 Mhz vs. 2.3 GHz), thus as we reported in Figure 15, it is more energy efficient, as well.

6.7. Hardware Resource Utilization

The flexible design of AxleDBs components allows for various compile time parameterization. For example, the sorter module can be configured for different depths, widths or ascending/descending orderings. Table IV shows the area reports, obtained by setting 16 Bytes for the key, and 48 Bytes for the $value$ field for the components. This compile time configuration covers all the requirements of the studied queries.

As it can be seen in Table IV, we observed that the data-parallel filtering and aggregation accelerators require a significant number of LUTs. The task-based join/groupby

Table III: Power Cons.

Component	Power(W)
clocks	**1.1**
logic	**0.77**
interfaces	**3.6**
signals	**1.1**
I/O	**1.01**
Acc/PDCS/RBAA DPC/FIFOs	**5.05**
leakage	**0.07**
Total	**12.7**
Total excluding interfaces	**9.1**

Table IV: Hardware Resource Utilization of AxleDB

Component	LUT	FF	BRAM	DSP
filter/aggr/arith	10396	2436	-	256
hash Join engine	13758	10623	724	-
sorter 128-node	148937	131730	-	-
merger (16-to-1)	33061	33840	33	-
index checker	2870	689	45	-
PDCS/RBAA DPC/FIFOs	2401	3884	26	-
SATA-3 ctrl	2018	2607	122	-
DDR-3 ctrl	12226	8329	1	-
PCI-3 ctrl	60671	62993	59	-
Total	283532	256490	1010	256
Virtex-7 usage (%)	65.7	30	68.6	7.1

modules require an extensive usage of hard memory blocks, because of a high amount of meta-data and the caching circuitry that is generated for providing higher performance. Also, to be able to support all the studied queries, the sorter is configured to be wide enough for running Q01, resulting in a large circuit that occupies around one-third of the FPGA. A 256-node sorter would occupy double this area and would cease to fit our FPGA along with all the other supported modules in AxleDB.

7. RELATED WORK

Previous studies have looked into designing efficient query processing engines, employing vector architectures [38], ASICs [4], GPUs [39] or hybrid [40], [41]. On the other hand, other approaches either used FPGAs statically [2], [24], or they leveraged dynamic reconfiguration to better fit the requirements of each query [42], [43], [44]. We differ from these works by following a static but programmable approach in query processing and data management, as we can support as many operations as we need, without requiring runtime reconfiguration. Industry has also invested on a few products, IBM Netezza [31] and XtreamData dbX [45], which offer full DBMS solutions.

We compared AxleDB with a set of state-of-the-art FPGA/ASIC-oriented query processing platforms: Ibex [6], Q100 [4], BlueDBM [36], [46], [47], and [46]. Ibex is a database storage engine that is equipped with a limited set of query processing operations, working directly with data inside SSD. Q100 proposes domain-specific database processors, but without supporting data management in off-chip storage. BlueDBM proposes a system architecture with flash-based storage and in-store processing capabilities, but it is not specialized for query processing. Sukhwani et al. [46] present an FPGA-based query processing engine that is attached to a DBMS via PCIe-3 with a data compression capability. Jaeyoung et. al. [47] present a smart SSD that incorporates it with memory and computing resources inside the SSD controller. [46] presents a query processing systems that efficiently uses partial dynamic reconfiguration capability of FPGAs to the on-the-fly query processing.

Table V lists the embedded accelerators for each of the studied platforms. AxleDB currently covers most of the necessary modules to run complex queries, although operations such as pattern matching or compression are not supported yet. On the other hand, as illustrated in Section 4, we proposed a novel and efficient accelerators for many important SQL query primitives. Our filtering and aggregation units are designed in an optimized way using modern HLS tools, our hash join engine is an enhancement of Ibex [6], and our sorter, based on the spatial sorter [25] is upgraded with

Table V: Comparing AxleDB with state of the art platforms, in terms of accelerators

	AxleDB	Ibex	Q100	BlueDBM	Sukhwani et al.	Jaeyoung et al.	Ziener et al.
filter	✓	✓	✓	×	✓	✓	✓
aggregation	✓	✓	✓	×	✓	✓	✓
hash join	✓	✓	×	✓	✓	×	✓
merge join	×	×	✓	×	×	×	✓
order by	✓	×	✓	×	✓	×	✓
DB indexing	✓	×	×	×	×	×	×
compression	×	×	×	×	✓	×	×

Table VI: Comparing AxleDB with state of the art platforms, in terms of features

	AxleDB	Ibex	Q100	BlueDBM	Sukhwani et al.	Jaeyoung et al.	Ziener et al.
Tech	FPGA	FPGA	ASIC	FPGA	FPGA	CPU	FPGA
COL/ROW	COL +ROW	COL	COL	——	COL	COL	COL +ROW
ISA	✓	×	✓	✓	✓	✓	✓
SSD	direct	direct	indirect	direct	indirect	direct	indirect
Cluster	×	×	×	✓	✓	×	×

extra LIMIT operation capabilities. Database indexing is embedded in the AxleDB and IBM Netezza, although Q100 is also equipped with a range partitioning method, as a preprocessing step for the ORDER BY operation, that could potentially be extended to serve for database indexing. Also, we summarized the technical characteristics of the given platforms in Table IV. More specifically, comparing AxleDB with:

–**Ibex**: we follow the similar approach to place the disk near the query processing units. However, Ibex can suffer from programmability capabilities in data management. Also, we propose a wider set of query accelerators, whereas Ibex is not equipped with a sorter accelerator or indexing capabilities. In Ibex, software fallbacks in some of the query accelerators such as hash join can diminish its throughput.

– **Q100**: our work differs mainly in micro-architecture and ISA design. Instructions of AxleDB are centered around data movement and initiating the accelerators, whereas Q100 has SQL-style instructions. The authors based their micro-architecture design on the sensitivity analysis of TPC-H queries. However, the off-chip bandwidth experiments have shown that query execution speeds are affected to a greater extent. In contrast, our platform is designed to maximize the available off-chip bandwidth with hardware indexing. This design effort is also complemented by explicit data movement instructions. Also, as Q100 is tailored as a composition of special purpose (ASIC) blocks, thus, extending the blocks can be an expensive process.

– **BlueDBM**: Its single node consists of flash storage, accelerator hardware in FPGA, flash controller and network interface. Data requests are sent from the host with minimum kernel overhead. Specialized accelerators process the data retrieved directly from flash storage, bypassing DRAM. AxleDB also supports different types of data movement, allows data streaming among the host, SSD, DDR-3, and accelerators. AxleDB is specialized for database query processing using an efficient set of query processing accelerators, whereas BlueDBM does not support the performing of such complex SQL queries. BlueDBM has a distributed structure that allows them to scale up more processing nodes, providing a larger address space. This capability is not yet supported

in AxleDB. However, the underlying architecture does not pose any difficulties for providing such functionality.
– **Sukhwani et al.** [46] presents an FPGA-based query processing engine. The engine is attached to a DBMS via PCIe-3. The supported functionality of the accelerator is filtering, join and sorting. To improve the throughput, the data is compressed by the host. Therefore, the decompression is the first step in the query processing pipeline on FPGA and processed queries are sent back in decompressed form. The authors do not provide any indexing mechanisms, and all queries that are sent to the accelerator require a full table scan. Also, the join and sorting units are not streaming based. Thus they use on-board DRAM for storing intermediate results. In the query pipeline, the filtering units always come before the join/sort units. Necessary data is read from DRAM of the host. Thus, this requires additional data movement from external storage to DRAM by either the host or the accelerator. In the AxleDB, the data movement and acceleration instructions allow processing blocks to execute in any order. Hence, it is possible to utilize a join/sort unit before filtering. AxleDB is designed to interface external memory directly and stores intermediate data in FPGA's memory. Also, AxleDB uses MinMax indexing to traverse data that resides in external SSD memory.
– **Jaeyoung et. al.** [47] presents a smart SSD that incorporates SSD storage with memory and computing resources inside the SSD controller. This design allows internal aggregate I/O bandwidth to be 5X higher than fastest SAS and SATA architectures. The Smart SSD architecture presented consists of embedded processors that are on the SSD host interface controller. They are coupled with DRAM and SRAM memories for intermediate data storage. NAND memory arrays inside SSDs allow parallel access. Contrary to Smart SSD, AxleDB is designed to work on an FPGA, and it is connected to an SSD via an SATA port. This allows the design of specialized accelerators on the FPGA rather than general purpose embedded processors. In two cases, smart SSDs can fail to exploit the advantages of its architecture, because these scenarios might not require extensive communication between the embedded processors and the SSD units. The first case is when the embedded processors require data communication between the host, and the utilization of the SSDs are very low. Next, the general purpose local memories of the embedded processors do not satisfy the memory requirements for the problem at hand. This might cause register spilling and decrease the performance. In these two cases, specialized accelerators can provide better results compared to general purpose embedded processors of the smart SSD.
– **Ziener et. al.** [44] presents a query processing platform that leverages the partial dynamic reconfiguration capability of FPGAs to better fit the requirements of each query on-the-fly. Query primitives such as filtering, aggregation, hash join, and sorter are gathered in a library while supporting both column- and row-oriented data store formats. However, this work does not follow a direct-SSD-coupled approach that diminishes the overall throughput, which can suffer from data offloading and partial reconfiguration overheads.

8. CONCLUSIONS AND FUTURE WORK

In this work, we demonstrated the design and implementation of AxleDB, a programmable query processing platform that couples efficient query-specific accelerators with COTS storage for providing better performance and energy efficiency compared to state-of-the-art software DBMS. In AxleDB, we targeted to decrease the execution time and to reduce the I/O overhead of data movement, simultaneously. To accelerate complex SQL queries, we proposed a diverse set of highly efficient query accelerators for aggregation, filtering, sorting, join and groupby operations. To reduce the I/O overheads, we used some techniques such as directly attaching SSD storage with column-oriented data to the processing units, and applying database indexing to dis-

card unnecessary data in the pre-processing time of the query. AxleDB was designed to be programmable by software through a set of special instructions, which enables flexibility in database acceleration in a novel way.
We observed that running a set of TPC-H queries, AxleDB can achieve query processing speedups ranging from 1.8X to 34.2X and more energy efficiency ranging from 2.8X to 62.1X, compared to the state-of-the-art DBMS, MonetDB, and PostgreSQL that run on a modern server.
Some of the promising future works are as follows: ***(i)*** developing the cluster version of AxleDB to support larger amounts of data, ***(ii)*** automating the hardware query plan, which is the translation process from SQL to AxleDB instruction set, ***(iii)*** equipping AxleDB with other effective database indexing methods, such as B-Trees to further optimize I/O transmission, and ***(iv)*** exploiting dynamic reconfiguration capability of FPGAs to better allocate the query-specific accelerators.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Program (FP7) (under the AXLE project GA number 318633), the Ministry of Economy and Competitiveness of Spain (under contract number TIN2015-65316-p), Turkish Ministry of Development TAM Project (number 2007K120610), and Bogazici University Scientific Projects (number 7060).

REFERENCES

[1] Balasubramonian, Rajeev and Chang, Jichuan and Manning, Troy and Moreno, Jaime H and Murphy, Richard and Nair, Ravi and Swanson, Steven. Near-data processing: Insights from a MICRO-46 Workshop. Micro, IEEE, volume=34, number=4, pages=36–42, 2014, IEEE.

[2] Casper, Jared and Olukotun, Kunle. Hardware acceleration of database operations. Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, pages=151–160, 2014, ACM.

[3] Chung, Eric S and Davis, John D and Lee, Jaewon. Linqits: Big data on little clients. ACM SIGARCH Computer Architecture News, volume=41, number=3, pages=261–272, 2013, ACM.

[4] Wu, Lisa and Lottarini, Andrea and Paine, Timothy K and Kim, Martha A and Ross, Kenneth A. Q100: the architecture and design of a database processing unit. ACM SIGPLAN Notices, volume=49, number=4, pages=255–268, 2014, ACM.

[5] Teradata. Teradata Appliance. 2015, "http://www.teradata.com".

[6] Woods, Louis and István, Zsolt and Alonso, Gustavo. Ibex: an intelligent storage engine with support for advanced SQL offloading. Proceedings of the VLDB Endowment, volume=7, number=11, pages=963–974, 2014, VLDB Endowment.

[7] Trzeciak, Artur. A Survey of Indexing Techniques for Object-Oriented Databases. 1997.

[8] PGSQL. PostgreSQL latest released version. 2015, "http://www.postgresql.org/".

[9] monetdb. MonetDB latest released version. 2015, "http://www.monetdb.org/".

[10] CStore-FDW. PostgreSQL Cstore Foreign Data Wrapper extension. 2015, "https://github.com/citusdata/cstore_fdw".

[11] Stonebraker, Mike and Abadi, Daniel J and Batkin, Adam and Chen, Xuedong and Cherniack, Mitch and Ferreira, Miguel and Lau, Edmond and Lin, Amerson and Madden, Sam and O'Neil, Elizabeth and others. C-store: a column-oriented DBMS. Proceedings of the 31st international conference on Very large data bases, pages=553–564, 2005, VLDB Endowmen.

[12] Woods, Louis and Eguro, Ken. Groundhog-a serial ata host bus adapter (hba) for fpgas. Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, pages=220–223, 2012, IEEE.

[13] TPC-H, TPC Benchmark H Standard Specification Revision 2.17.0, 2015, "http://www.tpc.org/tpch/spec/tpch2.17.0.pdf".

[14] Mueller, Rene and Teubner, Jens and Alonso, Gustavo. Glacier: a query-to-hardware compiler. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages=1159–1162, 2010, ACM.

[15] Xilinx. Vivado HLS tools. 2015, "http://www.xilinx.com/".

[16] Bluespec. Bluespec SystemVerilog Compiler. 2015, "http://www.bluespec.com/".

[17] Arcas-Abella, Oriol and Ndu, Geoffrey and Sonmez, Nehir and Ghasempour, Mohsen and Armejach, Adria and Navaridas, Javier and Song, Wei and Mawer, John and Cristal, Adrián and Luján, Mikel. An empirical evaluation of high-level synthesis languages and tools for database acceleration.Proceeding of 24th International Conference on Field Programmable Logic and Applications (FPL), pages=1–8, 2014, IEEE.

[18] Malazgirt, Gorker Alp and Sonmez, Nehir and Yurdakul, Arda and Cristal, Adrian and Unsal, Osman. High Level Synthesis Based Hardware Accelerator Design for Processing SQL Queries. Proceedings of the 12th FPGAworld Conference, 2015.

[19] Gorker Alp Malazgirt, Nehir Sönmez, Arda Yurdakul, Osman S. Unsal and Adrián Cristal. Accelerating Complete Decision Support Queries Through High-Level Synthesis Technology (Abstract Only). Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages = 277, 2015.

[20] Kocberber, Onur and Grot, Boris and Picorel, Javier and Falsafi, Babak and Lim, Kevin and Ranganathan, Parthasarathy. Meet the walkers: Accelerating index traversals for in-memory databases. Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pages=468–479, 2013, ACM.

[21] Salami, Behzad and Arcas-Abella, Oriol and Sonmez, Nehir. HATCH: Hash Table Caching in Hardware for Efficient Relational Join on FPGA. Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, pages=163–163, 2015, IEEE.

[22] Halstead, Robert J and Absalyamov, Ildar and Najjar, Walid A and Tsotras, Vassilis J. FPGA-based Multithreading for In-Memory Hash Joins, 2014.

[23] Becher, Andreas and Ziener, Daniel and Meyer-Wegener, Klaus and Teich, Jürgen. A co-design approach for accelerated SQL query processing via FPGA-based data filtering. Proceeding of International Conference on Field Programmable Technology, FPT, pages = 192–195, 2015.

[24] Parashar, Angshuman and Pellauer, Michael and Adler, Michael and Ahsan, Bushra and Crago, Neal and Lustig, Daniel and Pavlov, Vladimir and Zhai, Antonia and Gambhir, Mohit and Jaleel, Aamer and others. Triggered instructions: A control paradigm for spatially-programmed architectures. ACM SIGARCH Computer Architecture News, volume=41, number=3, pages=142–153, 2013, ACM.

[25] Arcas-Abella, Oriol and Ndu, Geoffrey and Sonmez, Nehir and Ghasempour, Mohsen and Armejach, Adria and Navaridas, Javier and Song, Wei and Mawer, John and Cristal, Adrián and Luján, Mikel. An empirical evaluation of high-level synthesis languages and tools for database acceleration. Field Programmable Logic and Applications (FPL), 2014 24th International Conference on. pages=1–8, 2014, IEEE.

[26] Oracle. Database indexing in Oracle. 2015, ”https://docs.oracle.com/database/121/ADMIN/indexes.htm”.

[27] Microsoft. Database indexing in SQL Server. 2015, ”https://technet.microsoft.com/en-us/library/ms190197(v=sql.105).aspx”.

[28] PGSQL-indexing. Database indexing in PostgreSQL. 2015, ”http://www.postgresql.org/docs/9.1/static/indexes.html”.

[29] PGSQL-BRIN. BRIN indexing method in PostgreSQL. 2015, ”http://www.postgresql.org/docs/devel/static/brin-intro.html”.

[30] Oracle-indexing. Storage Indexes in Oracle. 2015, ”http://www.oracle.com/technetwork/issue-archive/2011/11-may/o31exadata-354069.html”.

[31] IBM. IBM Netezza Data Warehouse Appliance. 2015. ”http: //www-01.ibm.com/software/data/netezza/”

[32] Manegold, Stefan. Private communication with MonetDB. 2015.

[33] PGSQL-Fixeddecimal. Fixeddecimal data type patch for postgreSQL. 2015, ”https://github.com/2ndQuadrant/fixeddecimal”.

[34] Terasic. ATA/SAS HSMC Card, 2015, ”https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=489”.

[35] Wu, Liang and Barker, Raymond J and Kim, Martha A and Ross, Kenneth A. Hardware partitioning for big data analytics.Micro, IEEE, volume=34, number=3, pages=109–119, 2014, IEEE.

[36] Jun, Sang-Woo and Liu, Ming and Lee, Sungjin and Hicks, Jamey and Ankcorn, John and King, Myron and Xu, Shuotao and others. BlueDBM: an appliance for big data analytics. Proceedings of the 42nd Annual International Symposium on Computer Architecture, pages=1–13, 2015, ACM.

[37] RAPL. Intel’s Running Average Power Limit (RAPL) energy meter, 2015, ”https://github.com/spandruvada/rapl_power_meter”.

[38] Hayes, Timothy and Palomar, Oscar and Unsal, Osman and Cristal, Adrian and Valero, Mateo. Vector extensions for decision support dbms acceleration. Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on. pages=166–176, 2012, IEEE.

[39] Power, Jason and Li, Yinan and Hill, Mark D. and Patel, Jignesh M. and Wood, David A. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-aggregate Queries. Proceedings of the 11th International Workshop on Data Management on New Hardware, pages = 11:1–11:8, 2015, ACM.

[40] He, Jiong and Lu, Mian and He, Bingsheng. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. Proceedings of the VLDB Endowment, volume=6, number=10, pages=889–900, 2013, VLDB Endowment.

[41] Arcas-Abella, Oriol and Armejach, Adrià and Hayes, Timothy and Malazgirt, Gorker Alp and Palomar, Oscar and Salami, Behzad and Sonmez, Nehir. Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory. Computing in Science & Engineering, volume=18, number=1, pages=80–87, year=2016, AIP Publishing.

[42] Becher, Andreas and Bauer, Florian and Ziener, Daniel and Teich, Jurgen. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. Field Programmable Logic and Applications (FPL), 2014 24th International Conference on. pages=1–8, year=2014, IEEE.

[43] Koch, Dirk and Torresen, Jim. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, pages=45–54, 2011, ACM.

[44] Ziener, Daniel and Bauer, Florian and Becher, Andreas and Dennl, Christopher and Meyer-Wegener, Klaus and Schürfeld, Ute and Teich, Jürgen and Vogt, Jörg-Stephan and Weber, Helmut. FPGA-Based Dynamically Reconfigurable SQL Query Processing. ACM Transactions on Reconfigurable Technology and Systems (TRETS), volume=9, number=4, pages=25, 2016, ACM.

[45] Scofield, Todd C and Delmerico, Jeffrey A and Chaudhary, Vipin and Valente, Geno. Xtremedata dbx: an FPGA-based data warehouse appliance. Computing in Science & Engineering, volume= 12, number= 4, pages=66–73, 2010, AIP Publishing.

[46] Sukhwani, Bharat and Min, Hong and Thoennes, Mathew and Dube, Parijat and Brezzo, Bernard and Asaad, Sameh and Dillenberger, Donna Eng. Database analytics: a reconfigurable-computing approach, Micro, IEEE, volume=34, number=1, pages=19–29, 2014, IEEE.

[47] Do, Jaeyoung and Kee, Yang-Suk and Patel, Jignesh M and Park, Chanik and Park, Kwanghyun and DeWitt, David J. Query processing on smart SSDs: opportunities and challenges. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pages=1221–1230, 2013, ACM.