ReSQM: Accelerating Database Operations Using ReRAM-Based Content Addressable Memory

Huize Li, *Graduate Student Member, IEEE*, Hai Jin, *Fellow, IEEE*, Long Zheng, *Member, IEEE*, and Xiaofei Liao, *Member, IEEE*

***Abstract*—The huge amount of data enforces great pressure on the processing efficiency of database systems. By leveraging the in-situ computing ability of emerging nonvolatile memory, processing-in-memory (PIM) technology shows great potential in accelerating database operations against traditional architectures without data movement overheads. In this article, we introduce ReSQM, a novel ReCAM-based accelerator, which can dramatically reduce the response time of database systems. The key novelty of ReSQM is that some commonly used database queries that would be otherwise processed inefficiently in previous studies can be in-situ accomplished with massively high parallelism by exploiting the PIM-enabled ReCAM array. ReSQM supports some typical database queries (such as SELECTION, SORT, and JOIN) effectively based on the limited computational mode of the ReCAM array. ReSQM is also equipped with a series of hardware-algorithm co-designs to maximize efficiency. We present a new data mapping mechanism that allows enjoying in-situ in-memory computations for SELECTION operating upon intermediate results. We also develop a count-based ReCAM-specific algorithm to enable the in-memory sorting without any row swapping. The relational comparisons are integrated for accelerating inequality join by making a few modifications to the ReCAM cells with negligible hardware overhead. The experimental results show that ReSQM can improve the (energy) efficiency by 611× (193×), 19× (17×), 59× (43×), and 307× (181×) in comparison to a 10-core Intel Xeon E5-2630v4 processor for SELECTION, SORT, equi-join, and inequality join, respectively. In contrast to state-of-the-art CMOS-based CAM, GPU, FPGA, NDP, and PIM solutions, ReSQM can also offer 2.2× 39× speedups.**

***Index Terms*—Content addressable memory (CAM), database query, nonvolatile memory, processing-in-memory (PIM).**

I. Introduction

IN THE big data era, modern enterprise data and Internet traffic have been exploding exponentially with a per-year growth amount that exceeds the total amount of data in the past years [1]. That exerts tremendous pressure on the existing database systems in a wide variety of data-intensive applications (such as biodiversity research [2]) for real-time data analytics demand, such that the response time of database operations must be much faster than ever before.

A wealth of the existing database systems are built upon CPU [3], [27], [32], [33], which is, however, difficult to satisfy the low-latency requirement due to its limited computational parallelism [8]. Alternatively, some efforts have been made in accelerating database operations with dedicated hardware. For instance, traditional CMOS-based content addressable memory (CAM) is developed as a coprocessor for CPU to achieve data-parallel computing for multiple database operations. However, it still relies on CPU to manage data transfer between CAM and main memory. In addition, due to the well-known scalability problem of the CMOS transistors, the computing ability of the CMOS-based CAM often suffers greatly in practice [9], [10]. Many studies leverage the massive parallelism of GPU [11], [12], [14], [15] (or FPGA [16], [18]) for (energy) efficiency improvement. Nevertheless, because of the separate computation-storage hierarchy by following the von Neumann architecture, these earlier studies suffer from the "memory wall" problem.

To address the above problem, near-data processing (NDP) integrates processing units into the memory or storage. Although significant data movement can be reduced for an NDP accelerator, they still suffer from challenges with the computing-ability-limited logic units in memory with considerable integration cost [19]–[21], [31]. Processing-in-memory (PIM) technology provides a promising way with the in-situ computing ability and massive parallelism. Sun *et al.* [22] presented a first PIM-enabled design to accelerate SQL query operations based on resistive random access memory (ReRAM). They exploit the bipolar structure characteristic of ReRAM crossbar and present a hybrid of columnwise and row-wise dot-product computations. Since the SELECTION operation contains some inherent comparison semantics that ReRAM does not support, they attach a simple peripheral scalar comparison unit to each row of ReRAM crossbar. This PIM-featured approach can offer the orders of energy efficiency over the traditional architecture, but its practicability still suffers. It is extremely difficult, if not impossible, for their approach to area-efficiently support complex but the important database operations, such as SORT and JOIN, which can be several million times comparisons than SELECTION in quantity for even a moderately sized database [4]. Yet, different

Manuscript received April 16, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported by the National Natural Science Foundation of China under Grant 61832006, Grant 61702201, Grant 61825202, and Grant 61929103. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Long Zheng.)*

The authors are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 1037, China (e-mail: huizeli@hust.edu.cn; hjin@hust.edu.cn; longzh@hust.edu.cn; xfliao@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2020.3012860

database operations involve different peripheral circuit layouts, making their design extraordinarily complex.

Recently, there emerges ReRAM-based content addressable memory (ReCAM), takes the best of both worlds of nonvolatile ReRAM [34], [35] and specialized CAM hardware with large capacity and PIM feature [24]. In addition to scalar comparison, ReCAM is also naturally capable of making the comparisons in a vector granularity, also known as vector–scalar comparison, *at a time* with higher parallelism. ReCAM is promising to enable in-situ in-memory computing to handle the database table for a wide variety of database operations efficiently. More importantly, the array structure of ReCAM can be intuitively regarded as a database table layout, making easy access to data and a fast mapping on ReCAM crossbars.

Nevertheless, exploiting ReCAM for accelerating database queries remains tremendously challenging. First, to support processing a database query, it is challenging to store and handle a lot of intermediate results. NVQuery [29] presents the first ReCAM-based accelerator for accelerating database operations. However, in order to obtain the final results of a query, NVQuery often relies on the main processor to process the intermediate results. Therefore, substantial data movements can be transferred between the processor and the ReCAM array, limiting the overall efficiency. Second, since ReCAM functions as both storage and processing units, the raw data of the database table in ReCAM must be consistent without data pollution for subsequent operations. This requirement may potentially suppress the efficiency of many database operations, such as the SORT that often involves (substantial) data reordering (if not carefully designed). Besides, vector–scalar comparison in ReCAM can compute only the equality between a given number and every element in a vector, affecting the applicability to handle some database operations, such as inequality join that needs to know the relativity [7].

In this article, we make the following contributions.

1) We identify that the existing PIM-based database-oriented accelerators can support a subset of database operations. Neither can support SELECTION, SORT, and JOIN queries simultaneously. Yet, these existing studies also typically rely on the main processor that assists a PIM architecture in handling a lot of intermediate results, which can become a bottleneck limiting the overall efficiency. To the best of our knowledge, ReSQM is the first ReCAM-based architecture that can process various database queries in memory effectively and efficiently without the assistance of a CPU processor.
2) We develop a series of hardware-algorithm co-designs to improve the efficiency of performance acceleration on different database operations. For SELECTION, we present a new data mapping mechanism that allows enjoying in-situ in-memory computations of the SELECTION query operating upon intermediate results for performance acceleration. For SORT, we develop a count-based ReCAM-specific algorithm to enable the in-memory sorting. For inequality-join, we make a slight modification to the basic ReCAM cell to support the relational comparison with negligible hardware overhead.
3) We conduct a comprehensive evaluation. We compare ReSQM with not only the traditional CPU-based, GPU-based, FPGA-based, and CMOS-based efforts but also the emerging NDP-enabled and PIM-enabled accelerators. Results show that ReSQM outperforms state of the art significantly.

The remainder of this article is organized as follows. Section II describes the background and motivation. Section III presents the architectural designs of ReSQM. Section IV shows the experimental results. Section V concludes the work.

II. Background and Motivation

A. Database Operations

In this article, we mainly focus on the relational database since it is widespread in the current mainstream market. In a relational database, those records with the same attributes are called *tuples*. In general, each tuple is distributed row by row to form a *table*. Each column of the table indicates an attribute of the table. In this article, we focus on three fundamental kernels of database queries as follows.

SELECTION: The selection query aims to choose tuples by querying a table via a restricted statement, which usually contains several arithmetic expressions connected with each other using various logical operators, such as AND, OR, NAND, and NXOR. The arithmetic operators used in the arithmetic expression may also get involved, e.g., "$+$," "$-$," "$\times$," "$=$," "$\neq$," $\leq$, $\geq$, "$>$," "$<$."

SORT: The sort query aims to reorder the tuples in an expected (e.g., ascending or descending) order according to some attributes.

JOIN: The join query aims to generate a new table using the Cartesian product of two relational attributes. In this article, we consider two typical join operations: 1) *equi-join* and 2) *inequality join*. The former indicates a join operation with the condition containing an equality operator of $=$. The latter represents a join condition with the inequality operators, e.g., $>$ and $<$.

B. ReCAM Basics

Fig. 1 illustrates the basics of ReCAM, which consists of a MASK register, a KEY register, an array of ReCAM bit-cells organized in a crossbar architecture, and TAG registers. The MASK register decides which columns will be selected to do read, write, and match operations. The KEY register stores a data word that will be used for a write or match operation. As shown in Fig. 1(a), a ReCAM bit-cell is organized with two transistors and two memristors (2T2R) elements with one bit line and one bit-not line. The match/word line of the ReCAM array is attached to a TAG register [Fig. 1(b)] in which each ReRAM array row is connected to a signal amplifier (SA) and a TAG latch. The TAG registers mark those matched rows that satisfy the condition of comparison. Unlike the row-oriented or column-oriented storage in a traditional memory [11], [28], the ReCAM crossbar is a natural fit to store the database table

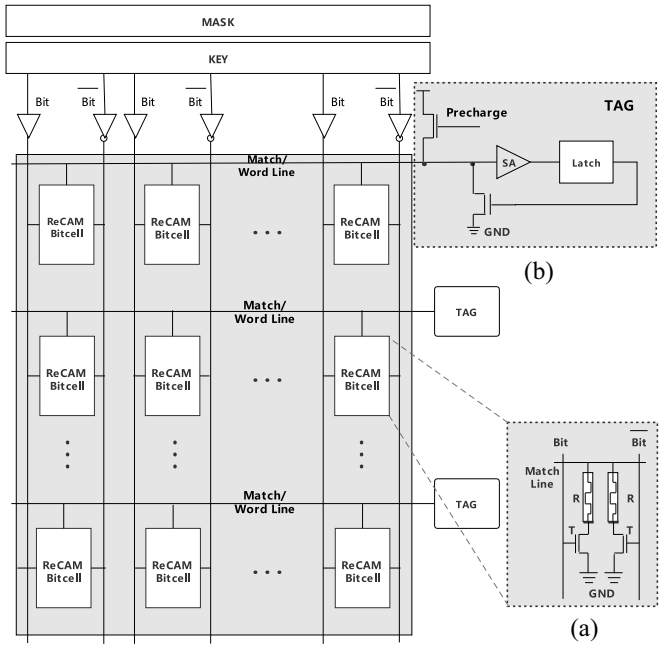


Fig. 1. Basics of the ReCAM array. (a) Sketch of ReCAM bitcell. (b) TAG register organization.

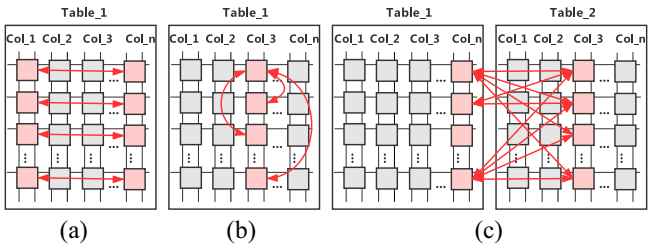


Fig. 2. Common computational patterns for (a) SELECTION, (b) SORT, and (c) JOIN.

with bit lines representing attributes and each match line showing a tuple. By using ReCAM, we can perform vector–scalar comparisons with massive parallelism.

As applied in [25] and [30], we follow to use the high-resistance state (HRS) to represents logic "1" (i.e., the switch-off state), while the low-resistance state (LRS) represents logic "0" (i.e., the switch-on state). Since a ReCAM cell often uses two memristive cells to represent one logic bit. We use the "10" of two memristive cells to represent the logic 1, and vice versa.

Vector–Scalar Comparison: Initially, a given scalar data that need to be compared is stored in the KEY register. All match lines are precharged with high voltage, while the KEY register was set on bit and bit-not lines. Note that the precharged signal and the signals operating upon the bit line and bit-not line of the KEY register are activated at the same time. The bit and bit-not lines of those columns that do not need to be compared are set to the low voltage by the MASK register. For each row (i.e., a vector element), if all selected bits match the given data, the corresponding precharged word line will keep high voltage that can be captured by the corresponding SA and also held in the TAG latch. Otherwise, if the mismatch of any one bit happens, leakage current will flow through that cell, and the voltage of the word line will drop off. Note that all per-row vector elements of selected columns against the scalar data can be compared in parallel and finished at one cycle.

C. Related Work

GPU and FPGA Acceleration: A lot of efforts have been put into speeding up database operations based on the traditional architectures, such as GPUs and FPGAs [11], [12], [14], [15], [18]. For instance, Schaa and Kaeli [12] pointed out that the Peripheral component interconnect express bus will also become a bottleneck on multiple GPUs unless the complete dataset can be placed in the memory of GPU. StoreGPU proposes to accelerate several hashing-based primitives for a distributed storage system [17]. By initializing the input data in the pinned host memory, StoreGPU protects the GPU driver from an extra memory copy with reduced data transfers. Asymmetric distributed shared memory [13] is proposed to maintain a shared logical memory space for reducing the amount of data movement between the host and the accelerator. An in-memory FPGA-based architecture is developed to accelerate table joins [16]. Compared with CPUs, these studies can provide superior results. Also, both GPU and FPGA acceleration of SQL operations can be designed with the flexibility that can deal with a larger set of SQL operations, types, and column/row sizes. However, currently, GPU and FPGA still suffer from the limited memory size such that they have to read/write through the host-system from/to SSD/HDD storage with I/O bottlenecks.

NDP and PIM Accelerators: Near-data computing integrates the processing units into storage or memory to reduce data access overhead [19]–[21], [31]. Although near-data computing can improve computing efficiency by reducing data movement, it still faces several challenges. Their processing ability of computing logic integrated into the storage and memory is quite limited, and also computational parallelism suffers. Integrating logic units into the stacking memory dies may also lead to a potentially high cost.

Sun *et al.* [22] presented the first PIM-enabled design based on ReRAM to accelerate SQL query operations. Due to the limited computational paradigm of the ReRAM array, this work can support only some operations of a SELECTION query. ReCAM has been widely used in many fields. Yavits *et al.* [23] replaced the last level cache with ReCAM as an associative processor. Kaplan *et al.* [25] leveraged ReCAM to accelerate the Smith–Waterman algorithm for DNA sequence alignment. To the best of our knowledge, NVQuery [29] is the most related ReCAM-based work specialized for accelerating database applications.

NVQuery presents a heterogeneous solution. It enables supporting some basic database operations based on ReCAM, such as nearest distance search, equi-join, and some bitwise operations. To obtain the final results of a query, NVQuery relies on the main processor to process the intermediate results. Therefore, an amount of data movement can be transferred between the processor and the ReCAM array, limiting the

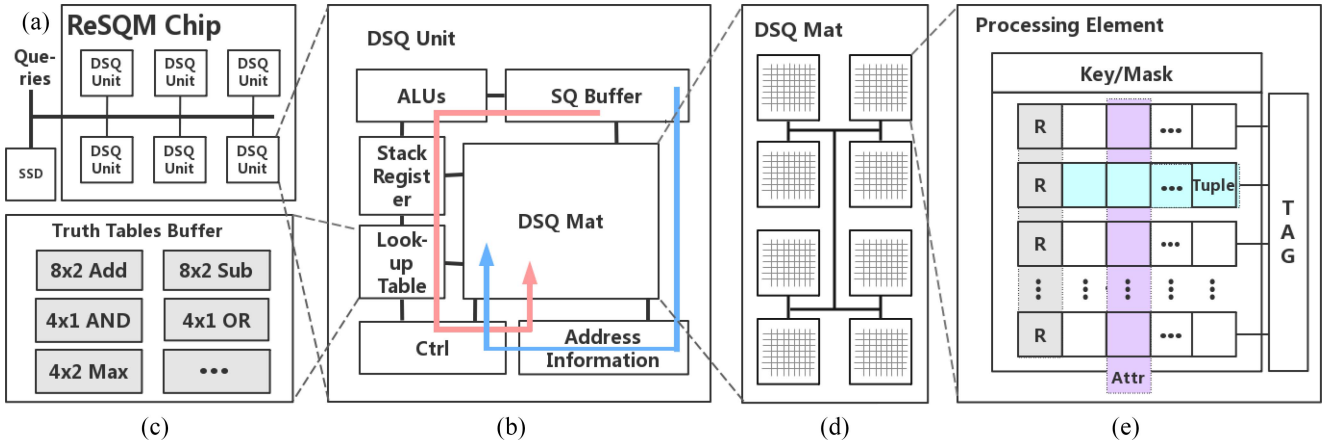


Fig. 3. Overview of ReSQM. (a) Layout of ReSQM chip. (b) Architecture of database structure query unit. (c) Truth table for supporting arithmetic and logical operators. (d) Interconnect of DSQ mat. (e) Table layout for the ReCAM array. The *pink arrow* shows the workflow for the SELECTION query. The blue arrow indicates the workflow of the SORT and JOIN queries.

overall efficiency. That is, particularly true and serious for handling a large database table. ReSQM differs from NVQuery in two ways: 1) we use an in-memory reserved region to buffer the intermediate results so as to perform operations between the intermediate results and the original data in memory and 2) each database structured query (DSQ) unit self-contains the arithmetic and logic units (ALUs) and a stack register, which can function to parse a restricted expression in SELECTION queries (without the assistance of the main processor), avoiding the substantial data transfer overheads.

We particularly note that the way of performing an addition operation in [29] is based on breaking it down into a serial of NOR operations, which is entirely different from ours that applies a lightweight and straightforward truth table (inspired from [24] and [25]). Although both are based on ReCAM, the architectures are different, and all algorithms that drive database operations are also different.

D. Motivation

Fig. 2 shows the computational patterns of SELECTION, SORT, and JOIN. We observe that database operations often involve many different practical demands that may be beyond the vector–scalar comparison pattern of the ReCAM array.

For example, in addition to comparison operators, the restricted expressions in SELECTION often involve many noncomparison operators operating upon the bitwise vector–vector computing, where only elements in the same row of two vectors are needed to take the computation [Fig. 2(a)]. Although SORT has the vector–scalar meta-operations [as shown in Fig. 2(b)], in the presence of the existing sorting algorithms (such as radix sort and merge-sort) they also involve the row swappings that ReCAM cannot support effectively. More importantly, the matching principle of ReCAM based on the leakage current mechanism can check only whether two elements are equal or not (i.e., equal comparison). However, most comparisons amongst database operations (such as inequality join) need to know more concerning which element is greater or smaller (i.e., the relational comparison).

III. ReSQM

Fig. 3(a) shows the overview of the ReSQM chip, which consists of multiple DSQ units connected through a bus that is used for receiving (sending) the (results of) queries from (to) users. Initially, we partition a database table into multiple slices such that each piece can fit into the DSQ unit. For handling an even larger table that cannot fit into the ReSQM's memory entirely, ReSQM can also work effectively by putting the large table in the solid-state-disk (SSD) storage, and send it to ReSQM in batches for processing.

In this section, we first elaborate on the architectural details of ReSQM, and then show the fundamental designs of accelerating different database queries.

A. Architecture

Fig. 3(b) shows the architecture of a DSQ unit that is the core of performing the execution for every received query. Note that ReSQM currently processes all queries serially. Supporting concurrent query execution can be considered an interesting future work. A DSQ unit contains some necessary components to support an effective query execution. Next, we discuss them as follows.

1) *Structured Query (SQ) Buffer:* It is mainly used to store the queries that will be processed by the DSQ unit. Yet, it can also be functioned to identify the type of a query, i.e., JOIN, SORT, or SELECTION.
2) *ALUs and Stack Register:* ReSQM includes some simple ALUs to convert a restricted expression used in SELECTION queries into a suffix one that can show the correct execution priority of the operators. Stack register stores the operands and operators during the expression parsing. ALUs and the stack register enable that ReSQM can work independently from CPU to accelerate database queries.
3) *Look-Up Table (LUT):* It is introduced to enable ReCAM to support basic arithmetic or logic operations based on the comparison paradigm of ReCAM. LUT stores the precalculated truth tables of basic instructions, as shown in Fig. 3(c).

TABLE I
COMMON ROW-WISE VECTOR–VECTOR INSTRUCTIONS USED IN DATABASE OPERATIONS

Instruction	Bit	Cycles
$C = A + B$	32	512
$B = A + B$	32	256
$C = A - B$	32	512
Row-wise max(A, B)	32	64
$C = A * B$	16	5184

4) *Address Information:* This is an address register that records and specifies which columns a particular attribute is stored in DSQ mat.
5) *Ctrl:* This is a local microcontroller that manages the components in the DSQ unit to perform the corresponding database operations and sends some control signals to the DSQ mat.
6) *DSQ Mat:* It is the main storage and computing component in the DSQ unit. It contains many processing elements that are connected through H-Tree, as shown in Fig. 3(d).
7) *Processing Element (PE):* Similar to prior work [24], we configure each PE with a ReRAM array size of 512 rows and 512 columns. Fig. 3(e) composes a sketch of data organization. We reserve the first 64 b (marked with "*R*") as a buffer to store the intermediate results (when necessary). The rest of the columns in PE are used to store the table.
8) *SSD:* An off-chip SSD is optionally used to store a large number of the results of a JOIN query when the ReCAM's on-chip memory is not sufficient.

In the ReSQM designs, we hold the argument that ReCAM arrays should function as both storage and computing units to eliminate the data movement between the processor and the memory. Based on this design philosophy, we next present how these key components are designed to accelerate SELECTION, SORT, and JOIN operations effectively and efficiently by exploiting ReCAM.

B. Accelerating SELECTION Queries

ReCAM can perform a variety of bitwise operations based on its vector–scalar comparison paradigm [24], [25]. To support SELECTION, NVQuery [29] proposes to transfer the operations of a SELECTION query into a series of bitwise operations, which can generate a large number of intermediate results. To obtain the final results of the query, NVQuery relies on the main processor to process these intermediate results. Consider the restricted expression in a SELECTION query contains a variety of bitwise operations, NVQuery often needs to transfer lots of intermediate results to the processor, degrading the overall efficiency.

To avoid the off-chip data transfers, we make the two core designs to perform a SELECTION query in memory. First, we reserve some memory spaces as *R* regions, as shown in Fig. 3(e), which will be used to store the intermediate results of SELECTION queries. Since the *R* region also has a computing ability, the intermediate results can also be computed with the original data for generating the next intermediate results, which can be further stored in the *R* region. In this way, ReSQM can get the final results of SELECTION queries by rational use of the *R* region. Second, we architect some ALUs and a stack register in each DSQ unit, as shown in Fig. 3(b). The ALUs will parse the restricted expression of a SELECTION query and obtain the correct execution of the restricted expression that will be stored in the stack register in the form of operands and operators. The processing of the restricted expressions is as follows.

In the beginning, two operands and one operator will be popped from the stack register and sent to the address information register and LUT, respectively. According to the truth table of this operator in LUT, the controller will generate suitable signals to the DSQ mat. With control signals applied to the memory address of these two operands recorded by the address information register, DSQ mat can calculate the results of these two vectors, and the results will be stored in the *R* region. After that, the stack register will pop an operator and an operand out to control the corresponding vector calculating with the intermediate results. When all operands have been processed, we can get the final result of the restricted expression in the *R* region. A logic 1 stored in the *R* region means that the value of the restricted expression is true on this tuple. Finally, ReSQM can get the results of this SELECTION query through a memory read request according to the value in the *R* region.

Example: Suppose two 32-b numbers "*A*" and "*B*" needs an addition. Next, we introduce how this simple operation is performed on ReSQM. This is a typical multibit addition case [23]–[25], which are often processed by transforming it into multiple single-bit additions. Then, we can use a truth table for the single-bit addition to process the operation. The procedure works as follows. First, the lowest bit of *A* and *B* will do a single-bit addition. The carry-out and result will be stored in the *R* region. Afterward, the carry-out will work as the carry-in and do an addition with the second-lowest bit of *A* and *B* to generate the next carry-out and result. This process is repeated until the highest bit of *A* and *B* is processed. Finally, the *R* region will hold the final result of "$A+B$". Note that the addition of the two elements in different rows is computed in parallel.

The multiplication can be considered multistep additions [23]. Row-wise max instruction is used to find a maximum number among the corresponding elements of two vectors in the same row.

In ReSQM, we perform the addition on the two 32-b vectors that operate on an 8-row truth table row by row. This thus takes $32 \times 8 \times 2 = 512$ cycles for a vector–vector addition. Other instructions are similar. Table I lists the instructions supported in ReSQM.

C. Accelerating SORT Queries

The sorting for some attribute columns based on the ReCAM array often needs first to perform the interrow comparisons and then reorder the attribute in different rows. However, the ReCAM array supports intercolumn comparison only. Also, the substantial amount of row swapping would

incur significant overheads in efficiency and energy consumption. Therefore, traditional sorting algorithms are often difficult to be applied for ReCAM. Imani *et al.* [29] used the difference between discharging currents to perform the nearest distance search. It has the key idea that for a given number, the closer the number on the match line is to the given number, the faster the current on the match line leaks. They use this way to find MIN and MAX results, which are a subset of SORT queries. However, as the number of records increases, this method will become challenging to differentiate data depending upon the discharging currents. Therefore, the method in [29] is hardly used for SORT queries, which are often operated upon millions of records in a table.

We present a count-based algorithm to support SORT queries on ReCAM effectively and efficiently. It can complete the ranking using the vector–scalar comparisons of ReCAM without any row swapping. The main idea is to construct a list of binary groups $\langle$digit, *cnt*$\rangle$, where digit is an element from an *attribute* column that needs to be sorted and *cnt* represents the repetition times of digit. These binary groups are generated serially according to the size of digit from the smallest to the largest. They will be stored in the *R* region [Fig. 3(e)] from the first line to the last line after the generation. Therefore, we can get a well-sorted attribute quickly by: 1) merely reading these binary groups in ascending order of rows and 2) then replicate the digit element *cnt* times. That is, the data in the *R* region can be treated as a well-sorted attribute that can be visible to users as the final result of the SORT query. Note that the number of binary groups might be large. Storing them in the *R* region can save not only many spaces but also avoid extra overheads of creating other data structures. Technically, applying the data structure of binary groups in ReSQM can also reduce a large number of writes on the ReCAM cells for boosting energy efficiency significantly.

The question is then how to generate a binary group according to the size of digit. Let us take the ascending order as an example. Fig. 4 shows the procedure of finding $\text{digit}_{\text{min}}$ and its corresponding *cnt* on attribute columns from the highest bit to lowest bit via `FindMinimumDigit`.

Initially, we clear all bits of the KEY register by 0. `FindMinimumDigit` seems like a filter algorithm, in which we step by step determine every bit of $\text{digit}_{\text{min}}$ and get rid of those elements that are definitely not $\text{digit}_{\text{min}}$ from the highest bit to the lowest bit. First, the MASK register will activate the highest bit to do a match operation between the highest bit of the KEY register and all elements of the attribute to be sorted. If some rows are tagged by the TAG register, it means the highest bit of $\text{digit}_{\text{min}}$ must be 0, and these unmatched rows will not be precharged any more because their highest bit is 1 and they can never be the smallest digit. If no row is tagged, this indicates that the highest bit of the $\text{digit}_{\text{min}}$ must be 1, and the highest bit of the KEY register will be set to 1. Thus, the precharge information will stay unchanged.

Afterward, DSQ Mat [as described in Fig. 3(b)] will activate the second highest bit of the MASK register to determine the second highest bit of the $\text{digit}_{\text{min}}$ to be 0 or 1. DSQ Mat will repeat the same procedure until the lowest bit is matched. After this phase, the number in the KEY register has stored

```
1  Procedure FindMinimumDigit(SortedColumn Vector)
2      set all bits in the KEY register by '0'
3      pre-charge all rows
4      i ← Sizeof(Vector)
5      while i >= 0 do
6          validate the i-th bit in the MASK register
7          if no row matched then
8              write the i-th bit in the KEY register by '1'
9              pre-charge information doesn't change
10         else
11             pre-charge only the matched rows latched in the
                TAGs
12         i = i - 1
13     return ⟨key, Number of the matched rows⟩
```

Fig. 4. Finding a minimum *digit* and its count.

every bit of the $\text{digit}_{\text{min}}$. All the rows matched can be considered as $\text{digit}_{\text{min}}$, and their number indicates the *cnt* of $\text{digit}_{\text{min}}$. Through `Sizeof(Vector)` (e.g., 32 in this article) cycles, we can find a minimum digit and its corresponding count. All the rows matched to this $\text{digit}_{\text{min}}$ will not be precharged so as to find the next $\text{digit}_{\text{min}}$. Note that the *i*th minimum digit can be easily found by disabling matching the $(i-1)$th minimum digit.

For every sorting queries, the same operations are performed upon the KEY and MASK registers of all processing elements (PEs). Each PE can execute `FindMinimumDigit` in parallel under the control of the KEY and the MASK registers. Since each PE returns a cnt_i of the same digit. By simply adding all cnt_i in the ALU, the *cnt* of $\text{digit}_{\text{min}}$ can be obtained then.

Computational Complexity: The computational complexity of our sorting procedure is $O(NM)$, where N is the number of elements and M is the number of elements with duplicates. Suppose all the elements are unique, the worst complexity will be $O(N^2)$, which can be finished in N cycles under ReSQM.

D. Accelerating JOIN Queries

Compared with SELECTION and SORT operations, the results of a JOIN query can be too large and might exceed the memory size of ReSQM. In this case, ReSQM enables to optionally store the massive results of a JOIN query into an off-chip SSD instead of the *R* region. Once a join-induced matching (a part of final results) is found, it can be (optionally) transferred to the SSD (if necessary). Note that these off-chip data transfers can be conducted in an overlapping fashion with the normal executions. Hence, their impact of off-chip data movements can be mitigated as well.

Equi-Join: Imani *et al.* [29] used a so-called "exact search" mode to support equi-joins based on the LUTs. For Equi-Join, the number of table lookups can often be thousands of times that of SELECTION. Once the LUT is occupied by some operations, they often require considerable overheads to finish the operations, since the frequent switching of control signals has occurred. The LUT becomes a bottleneck for equi-join. ReSQM copes with this issue by performing a data reading in advance to use the vector–scalar comparison ability of ReCAM, without the assistant of LUTs for equi-join.

The equi-join implementation of ReSQM can be as simple as performing some per-tuple scalar-vector comparisons. Suppose we hold two attribute columns A and B from two tables. All records of A will be read out in turn and sent to the KEY register to compare with B simultaneously. All matched rows tagged by the TAG register can be part of the results of the equi-join of A and B. Note that the intermediate results of the equi-join between B and each element of A can be optionally written into the SSD. Note that although our method introduces extra read operations, ReSQM can still preserve the efficiency for the following reason. The reading of A can run while the processing of B in an overlapping manner since A and B are stored in different tables.

Inequality Join: Unlike SELECTION, SORT, and equi-join that performs equal comparisons, which have a good fit for the ReCAM computational paradigm, inequality join involves the relational comparison. To enable relational comparison, we make a slight modification to the ReCAM bit-cell organization with respect to the current leakage mechanism.

Fig. 5 shows the modified structure of ReCAM bit-cell that we add a TAG-G register to each row of ReCAM. The main idea is to architect an extra TAG to capture the direction of leakage current when dismatched. We architect a TAG-G register between all memristors of the bit line and the ground wire to detect the potential leakage current. When ReSQM performs SELECTION, SORT, and equi-join queries, the switch controller (SC) in TAG-G will be in the switch-on state to make the SA and the latch invalid. The SC will be in the switch-off state if inequality join query is under processing. Suppose the KEY register stores 1 and ReCAM bit-cell stores 0, leakage current will flow through the switch-on memristor on the bit line to the TAG-G since the memristor on the bit-not line is in the switch-off state in this case. Then, both of TAG-M and TAG-G will store a logic 1. The modified bit-cell may have three cases: 1) no leakage current occurs. Only the TAG-M will capture the voltage, indicating the equality; 2) the TAG-G and the TAG-M both capture a leakage current, indicating a greater data in the KEY register; and 3) both TAG-M and TAG-G do not capture a signal, meaning a smaller-than relation.

Note that we architect TAG-G based on the off-the-shelf TAG architecture. The timing correctness of its SA and latch controller has been demonstrated in previous studies [23]–[25]. The working mechanism of SA and latch is easy to understand. Both SA and latch have an internal resistance. Therefore, the current often prefers to be leaked out of the memristor preferentially. If and only if the current cannot flow through the memristor, the match line will then hold a high voltage. In this case, the SA and latch will start working. The activated timing of the precharge signal and the signals operating upon the KEY register is the key for the correctness of the TAG circuit. We also note that these signals are activated at the same time, and thus, the correctness can be ensured.

The basic principle of performing the relational comparison between two data based on the modified bit-cells works as follows. We can perform a bitwise comparison from the highest bit to the lowest bit of two data. The relational comparison between their highest bits can directly indicate their relationship of the size. If their highest bits are equal, we can iteratively compare their subsequent bits from the high to the low bit until a nonequal relation is found. Otherwise, the two data are essentially equal. Similar to equi-join, inequality join also performs the meta-operation of scalar-vector comparison.

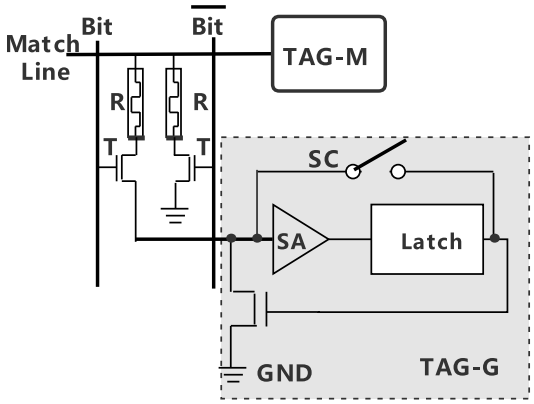


Fig. 5. Modified ReCAM bitcell organization. Each TAG-G is shared.

TABLE II
WORKLOAD CHARACTERISTICS WHERE $m = 1, 4, 8, 16$

Para.	Value
size of key	4 bytes
size of M	$4m\times10^6$ tuples
size of N	16×10^6 tuples
total size M	$m\times128$ MB
total size N	512 MB

Zhao *et al.* [30] presented a relational comparator for the random forest. They divide the original bit lines and bit-not lines into the two separate ReRAM arrays. Through precharging the two arrays individually, the relativity of two data can be obtained by computing the voltage difference between the two arrays. Unfortunately, this approach suffers in accelerating database operations, which involve not only the relational comparisons but also a large number of equal comparisons (as in SELECTION and SORT operations) or even noncomparison operations. Their separate architecture, in many cases, might double the overheads of those database operations since at least double rows need to be precharged. In contrast, ReSQM adds only a neat and cheap TAG register attached to each row with nearly negligible modification to the ReCAM architecture without sacrificing any potential parallelism of the ReCAM array.

Computational Complexity: The JOIN query often needs $O(NM)$ matchings, where N and M represent the lengths of two attribute columns A and B, which can be accelerated in N cycles under ReSQM if A is read to match B.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

Workloads: Table II lists the workloads used for ReSQM. We used the GNU library [11] to create two tables M and N. Both have nine attributes, among which the key has 4 B, and two attributes are the 2-B integers used for the 16-b multiplication while the rest is the 4-B integers. The key attribute was

TABLE III
ATTRIBUTE DISTRIBUTION OF THE TABLE *M* AT DIFFERENT SIZES, AND THE TABLE *N*

table	4-byte attributes			2-byte attributes	
	Uniform	Bernoulli	Gaussian	Uniform	Gaussian
M@4M	$(-10^4, 10^4)$	(0, 1)	$\sigma = 200$	$(-2^{15}, 2^{15})$	σ=1 × 10^4
M@16M	$(-10^6, 10^6)$	(0, 1)	$\sigma = 20$	$(-2^{10}, 2^{10})$	$\sigma = 100$
M@32M	$(-10^8, 10^8)$	(0, 1)	$\sigma = 5000$	$(-2^{20}, 2^{20})$	$\sigma = 40$
M@64M	$(-10^{10}, 10^{10})$	(0, 1)	σ=2 × 10^4	$(-2^5, 2^5)$	$\sigma = 7000$
N@16M	$(-10^7, 10^7)$	(0, 1)	$\sigma = 1500$	$(-2^{13}, 2^{13})$	$\sigma = 350$

marked as attr_0, while others are marked from attr_1 to attr_8 in turn. As shown in Table III, these attributes are generated based on the Bernoulli, uniform, and Gaussian distributions. In particular, we generate table *M* at four different scales for sensitivity study.

Measurement: We evaluate ReSQM by two metrics: 1) *response time* and 2) *energy consumption*. All results for ReSQM and the baseline are obtained by an average of running ten different queries for each query type. SELECTION and SORT are performed at the table *M* with all the four scales. Equi-join and inequality join are performed on table *N* and four scales of table *M*. We list each typical query as follows.

1) *SELECTION (SE):* Select attr_0, attr_2, attr_5, and attr_8 from the table *M* where $2 \times (\text{attr}_5 + \text{attr}_6 - \text{attr}_7) > 3000$ AND $(\text{attr}_2 - \text{attr}_4 - \text{attr}_1) < 500$ OR $4 \times (\text{attr}_8 + \text{attr}_5) + 5 \times (\text{attr}_7 - \text{attr}_6) > 1000$.
2) *SORT (SO):* Select attr_3 from the table *M* order by attr_3.
3) *Equi-Join (EJ):* Select $M.\text{attr}_2$, $M.\text{attr}_4$, $N.\text{attr}_0$, and $N.\text{attr}_3$ from the tables *M* and *N* where $M.\text{attr}_2 = N.\text{attr}_3$.
4) *Inequality Join (IJ):* Select $M.\text{attr}_5$, $M.\text{attr}_3$, $N.\text{attr}_3$, and $N.\text{attr}_1$ from the tables *M* and *N* where $M.\text{attr}_3 > N.\text{attr}_1$.

Cycle-Accurate Simulation: We use a cycle-accurate simulator in which the underlying mathematical model constraints have been proved to ensure the correctness and accuracy for program executions [23]. ReSQM applies this with a three-step simulation [23] for the ReCAM hardware. The first step is data mapping. ReSQM can work as a memory. The two tables *M* and *N* will be written into the DSQ Mat of all the DSQ units. Their attribute locations are also recorded in the address information register. The second step is to decompose a database query into a serial of arithmetics and data communication operations. This step is managed by the controller in the DSQ unit. Consider the SELECTION query as an example. The original query statement will be parsed as arithmetic expressions shown in Table I. Finally, these arithmetic operations will be converted into a serial of ReCAM atomic operations, such as read, write, and comparison. Through looking up Table I, we can hence obtain the running time of each query accurately. The simulation for SORT and JOIN needs only the first and the last steps since we perform their algorithmic operations straightforward based on the ReCAM atomic logic rather than arithmetic operations.

ReSQM Configurations: ReSQM runs at 1 GHz with 12 DSQ units. We use the SPICE simulator to obtain the energy consumption, area parameters, and performance of ReSQM. Each DSQ unit has 437.5-MB memory and 200-W power at full capacity. The area of each DSQ mat is 103.7 mm^2 and each ReCAM array takes 0.0034 mm^2. The read and write operations are atomic, and their latency is 8.31 and 17.42 ns, respectively. The matching operation is also atomic during the computation. Its latency is 1 ns, an inverse of the frequency. In general, the read and write operations are more expensive than the match operation, and hence we implement our algorithms by using read and write operations at a minimum level. Taking SORT as an example, we do not use any reads and use writes only when the binary groups need to be stored.

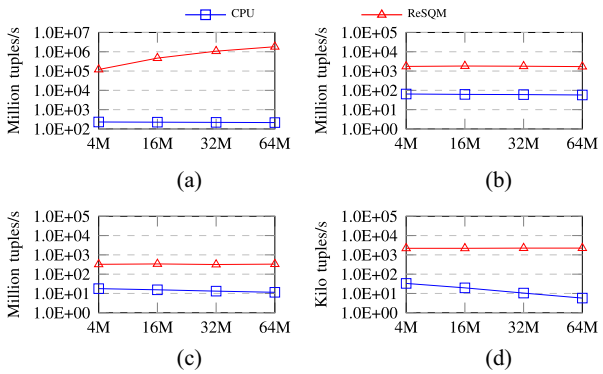


Fig. 6. Throughput (tuples processed per second) of ReSQM when hanlding all the four database operations on the table *M* at different sizes. (a) SE. (b) SO. (c) EJ. (d) IJ.

In this article, we conservatively set the size of the KEY register to 32 b for careful consideration of supporting the bitwise operations that are frequently used in SELECTION. As discussed in Table I, performing bitwise operations is often sensitive to the bit number. As the size of the KEY register increases, the number of cycles required can significantly increase. In this case, the KEY register is scanned and processed one bit at a time in a bit-serial manner [23]–[25]. In actuality, the size of the KEY register can be still set to 256-b if only SORT and/or JOIN queries are considered.

We evaluate ReSQM against a baseline with a 10-core Intel Xeon E5-2630v4 CPU@2.2 GHz, 25-MB Cache, 68.3 GB/s, and 85-W TDP. We run SQLite selection, radix sort [14], sort-merge join [27], and inequality join on PostgreSQLv9.4 [5] for our baseline comparison. We use RAPL [26] to measure the energy of the CPU. ReSQM is a PIM accelerator and all tables and the results of SELECTION and SORT are stored in the memory. The baseline also stored all tables and the results of SELECTION and SORT in the memory. The JOIN results of ReSQM and baseline are both stored in the off-chip SSD. The SSD card connected to ReSQM is the same as that to CPU with a size of 480 GB. The SSD interface is based on SATA3. Its read and write speeds are 562 and 420 MB/s, respectively. To make apple-to-apple comparisons, we benchmark ReSQM against the CPU baselines using the same workloads and the same benchmarks. For preserving fairness, the loading time of the original data from the disk to the memory is not counted.

B. Overall Efficiency

We first evaluate the speedup and energy consumption of ReSQM against the CPU baseline for SELECTION (SE), SORT (SO), Equi-Join (EJ), and Inequality Join (IJ).

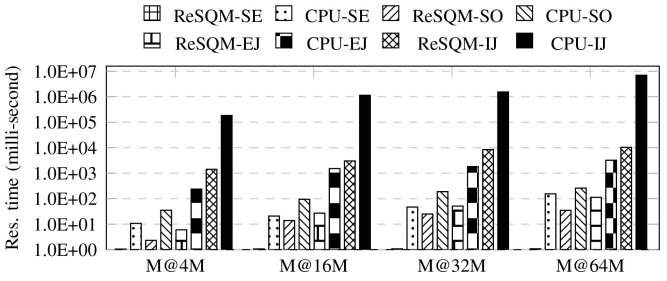


Fig. 7. Response time of ReSQM against CPU for all the four database operations on the table *M* at different sizes.

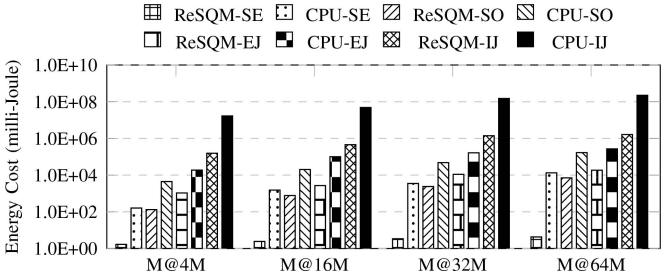


Fig. 8. Energy consumption of ReSQM against CPU for all the four database operations on the table *M* at different sizes.

Throughput: Fig. 6 shows the throughput between ReSQM and CPU-based platform. For the SELECTION query, we see that ReSQM can achieve 162G tuples per second, while the CPU can only process 359M tuples per second for the table M@4M. In particular, when the table is scaled to 64M, ReSQM can continue improving the throughput to up to 1083G tuples per second while the throughput of CPU stays immutable. For the SORT query, ReSQM has an average throughput of 1096M tuples per second while CPU has an average throughput of 81M tuples per second only for the four scales of the table *M*. For the equi-join query, ReSQM has an average throughput of 513M tuples per second while CPU has an average throughput of 13M tuples per second only. For the inequality join query, ReSQM has an average throughput of 3.5M tuples per second. The CPU has a typically small throughput of 15.2K tuples per second for table M@4M. When the table size is at 64M, the throughput of CPU becomes worse by 7.6K tuples per second.

Speedup: Fig. 7 shows the speedup results. Overall, ReSQM significantly outperforms CPU for all database operations. For example, for the table M@4M, ReSQM can accelerate SE, SO, EJ, and IJ in 0.025 ms, 3.65 ms, 7.8 ms, 1.15 s, while CPU complete them by 11.13 ms, 49.37 ms, 319.5 ms, 263.6 s, yielding the speedups of 445×, 13×, 41×, and 229×, respectively. More importantly, ReSQM shows better scalability than CPU as the data size increases. Taking IJ as an example, its response time can be increased from 17.9 to 8 469 s when the table *M* varies from 4M to 64M, while ReSQM keeps the response time between 1.15 and 17.9 s with significant improvement by the two to three orders of magnitude. That yields the speedups of 721×, 25×, 95×, and 471× for the table M@64M.

Energy Efficiency: Fig. 8 shows the energy results further. We can see that ReSQM can complete all the database operations with far fewer energy consumption against the baseline due to the substantial reduction of data movement. For the table@4M, ReSQM can reduce energy consumption by 164×, 12×, 21×, and 114× for SE, SO, EJ, and IJ, respectively. A better scalability of ReSQM further reduces them into 239×, 23×, 55×, and 237× for the table M@64M.

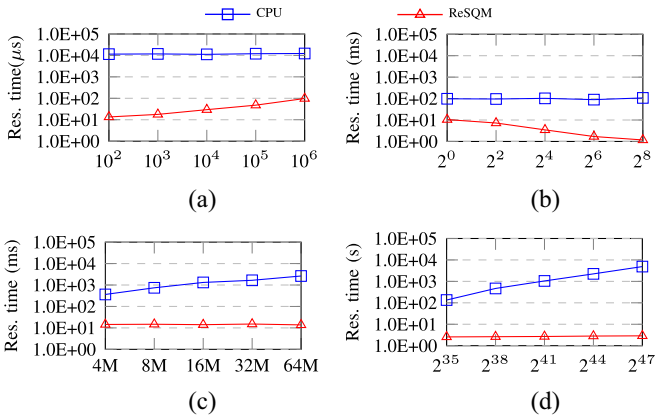


Fig. 9. Response time of ReSQM against CPU with varying query result sizes. All results are obtained on M@16M. (a) SE. (b) SO. (c) EJ. (d) IJ.

C. Systematic Impact of Query Result Size

We further show the systematic impact of ReSQM when the result size of query increases for SE, SO, EJ, and IJ.

SE: Fig. 9(a) characterizes the performance of ReSQM for SE against CPU. We can see that the CPU seems to be insensitive to the query result size, while ReSQM is sensitive. The reason is below. In a CPU architecture, no matter how many tuples are matched with the restricted expression, it always needs to load all tuples from memory to cache for a global analysis, thereby yielding a relatively stable performance. On the contrary, ReSQM performs the in-situ computation of restricted expression with only those columns that need comparisons computed. Despite the rising tendency in response time, ReSQM still has a faster response speed than baseline due to fewer data movements.

SO: Fig. 9(b) characterizes the performance of ReSQM against radix sort on CPU with different repetition times of every unique element on an attribute that needs to be sorted. As the repetition times increase, results show that the response time of CPU maintains a stable level while that of ReSQM is reduced significantly. Radix sort is easy to understand since it needs to compare every element in each round. In ReSQM, the same elements are reduced into one unique binary tuple such that we can access only once to this unique binary tuple to remove the redundant accesses.

EJ: Fig. 9(c) shows the response time of EJ, for which CPU has an increasing overhead while ReSQM's is degraded. The reason is as follows. CPU-based EJ includes a sort-merge two-step approach. As discussed above, the performance of sorting keeps stable. For the merging operation, a large size of results often implies more than row-wise comparisons, thereby leading to longer response time. The case is completely different in ReSQM. It has no sort step. Also, the EJ is a natural fit

TABLE IV
COMPARISONS BETWEEN THE ORIGINAL RECAM ARRAY AND OUR MODIFIED ARRAY

Para.	Original	Ours
Match latency of SE	1ns	1ns
Match latency of SO	1ns	1ns
Match latency of EJ	1ns	1ns
Match latency of IJ	1ns	1.1ns
Power of DSQ Mat	200W	200W
Area of DSQ Mat	$101.4mm^2$	$103.7mm^2$

TABLE V
OVERHEAD BREAKDOWN

Operations	Selection		Sort		Join	
Components	CTRL	CAM	CTRL	CAM	CTRL	CAM
Ratio	16.23%	83.77%	1.72%	98.28%	2.44%	97.56%

TABLE VI
ENERGY BREAKDOWN

Components	ReCAM		Others
	Leakage	Dynamic	
Ratio	8.14%	89.79%	2.07%

TABLE VII
AREA BREAKDOWN

	Total Area: $115mm^2$				
	Peripheral Circuits			DSQ Mat	
	Buffers	ALU	Controller	H-tree	CAM
Ratio	7.57%	1.59%	2.41%	0.04%	88.39

TABLE VIII
PERFORMANCE OF RESQM AGAINST GPU, FPGA, NDP, AND PIM PLATFORMS (NORMALIZED TO CPU PLATFORM)

	SELECTION	SORT	Equi-Join	Inequality Join
GPU	36× [11]	0.6× [14]	4.6× [15]	7.8× [6]
FPGA [16]	41×	8.7×	6.4×	-
NDP	9.3× [31]	-	-	-
NVQuery [29]	28.2×	-	8.7×	-
ReSQM	611×	19×	59×	307×

for vector–scalar comparisons. More comparisons involved in a large query result size can be well parallelized by exploiting the massive parallelism of the ReCAM array.

IJ: Fig. 9(d) shows the response time of IJ. By rearchitecting the ReCAM bit-cell, ReSQM also exposes the massive parallelism of ReCAM to handle the relational vector–scalar comparisons. Thus, the performance of ReSQM against CPU for processing IJ shows a similar variation trend as processing EJ.

D. Overheads and Breakdown

TAG-G Overheads: We evaluate the overheads of the TAG-G register. Table IV shows the latency and energy consumption of the match operations between the original ReCAM array and our modified array. We see that only the match latency of the inequality join is 0.1 ns longer than the original array. The latency for other database operations is the same as the original array. The reason is apparent that TAG-G is not used when performing SELECTION, SORT, and equi-join, and their performances are not affected by this modification. The energy consumption for all database operations is not influenced either by this modification because the TAG-G makes full use of the existing leakage current mechanism, rather than architects the new hardware components. Finally, we also see that the area of DSQ Mat with TAG-G is 103.7 mm^2, introducing only an extra 2.3 mm^2 against the original array without TAG-G.

Controller Overheads: We further investigate the controller overheads when handling SELECTION, SORT, and JOIN queries, respectively. Table V depicts the results. "CAM" indicates the overhead from the DSQ Mat. We can see that the overhead of the controller takes 16.23% of the total overheads for the SELECTION query. The controller can be as small as 1.72% and 2.44% in the SORT and JOIN queries, respectively.

The reasons for the low overheads of the controllers are simple. The execution logic of the DSQ Mat is driven by the KEY and MASK registers, which are dependent on the data transferred from the DSQ unit, further dependent on the controllers. For the SORT and JOIN queries, the MASK and KEY registers work in a regular way. For example, in `FindMinimumDigit`, the MASK register is activated bit by bit from the highest bit to the lowest bit, and the KEY register is initially set to 0. Therefore, with just one signal from the controller, the MASK register can work 32 times and find the $\text{digit}_{\text{min}}$ and its count. However, for the SELECTION query, the situation is different. The execution of each arithmetic operation needs control signals from the LUT. Therefore, one signal from the controller can manage the write of only one row to the KEY register.

Energy and Area Breakdown: We also investigate the energy consumption and area of each component in ReSQM. In Table VI, we can see that the ReCAM array consumes most (97.93%) of energy, among which dynamic computations (replying on the precharging, KEY register, MASK register, etc.) occupy 89.79% of energy consumption while leakage current takes 8.14% energy. The other components beyond the DSQ mat cost only 2.07% energy. Table VII further shows the area breakdown of ReSQM. The ReCAM array occupies 88.39% of the total area, with the ratio of H-tree is 0.04%. All buffers, ALUs, and microcontrollers have only 7.57%, 1.59%, and 2.41% area, respectively. By adding small add-on peripheral circuits, ReSQM functions well as a promising-in-memory device to accelerate database operations.

E. Compared With Other Platforms

We finally evaluate ReSQM against some state-of-the-art GPU, FPGA, NDP, PIM, and CMOS-CAM-based efforts. Note that some of these studies may support a part of four database operations involved in this work.

For GPU, we use an NVIDIA GTX1080@1733 MHz, 2560 Cuda Cores, 2-MB shared L2 Cache, 8-GB Graphic Memory, and 180-W TDP. The SELECTION algorithm is introduced from [11], the SORT algorithm from [14], Equi-join from [15], and inequality join from [6]. For FPGA, we use the architecture and algorithms from [16] for SELECTION, SORT, and equi-join queries. The NDP baseline is from [31] for SELECTION only. As for the PIM baseline, we select

NVQuery [29] for SELECTION and equi-join queries. To ensure fairness, we evaluate ReSQM against these baselines by running the same benchmarks on the same workloads.

Performance Comparisons and Analysis: Table VIII shows the performance results for GPU, FPGA, NDP, and PIM platforms. We can see that ReSQM shows the best performance by the speedups of 15×, 2.2×, 6.8×, and 39× over the better performer among GPU, FPGA, NDP, and PIM platforms for SE, SO, EJ, and IJ, respectively. For SELECTION, the NDP accelerator offers the worst acceleration effect compared with other platforms. This is because, for a large table, [31] relies on a CPU to process lots of operators and intermediate results. Thus, the data transfer bottleneck limits the overall efficiency. Compared with NVQuery [29], ReSQM offers more than 30× speedup, due to the reduced number of intermediate result transfers. Since SELECTION is as simple as being with good data parallelism, GPU and FPGA platforms show the superior results over NVQuery for all database tables.

Note that ReSQM on SO shows a relatively less speedup than those on SE, EJ, and IJ due to the underutilization of ReCAM bit-cells. Actually, only 5% of bit-cells are used for SO in ReSQM. The rest (unrelated to a sorting attribute) is aggressively disabled for correctness. On accelerating SO faster by fully utilizing ReCAM resources better, we leave it as future work. For equi-join, which represents higher complexities than SELECTION, we see that NVQuery becomes superior against GPU and FPGA. Without the lookup overheads of LUT, ReSQM offers more than 6× speedup over NVQuery. For inequality join, only GPU and ReSQM can support it currently. However, we still find that ReSQM outperforms GPU by 39×, due to the in-situ computing ability and massive parallelism of the ReCAM array.

For the CMOS-based CAM, it often suffers from the severe scalability issue with the limited dataset supported. To facilitate comparison with the existing work, we use similar workloads to [9] by performing SO on a 40 000-tuple table, and running EJ and IJ on two tables with 20 000 tuples and 40 000 tuples, respectively. For SO, EJ, and IJ, CMOS-based CAM can offer the speedups of 1.59×, 7.3×, and 11.2× against CPU, while our accelerator offers 7.7×, 21×, and 136×.

F. Discussion

So far, using ReCAM to handle string types has some difficulties with many challenges faced, particularly lack of an effective data mapping: 1) using a fixed size to represent a character is often difficult, if not impossible, to support an arbitrary-length string. Supposing we use 26 English letters as a collection to generate strings, so each character is represented by 5 B, one row of the ReCAM array can often support a maximum of 50-character string only; 2) using multilevel cells (MLCs) can mitigate the above issue to support a relatively long string. However, this needs a strict MLC production process and also introduces a precision problem; and 3) using a fixed size of the ReCAM array size to support irregular strings is also difficult, which needs a valid tradeoff between computational parallelism and storage efficiency.

This work is just small-step research of using ReCAM to accelerate some database queries. Although supporting strings remains an open question, we believe that ReSQM still has addressed several critical challenges in this timely topic and would facilitate the subsequent research of handling strings effectively and efficiently.

V. Conclusion

This article identified a spectrum of comparison semantics in the relational database operations. We introduce ReSQM, a novel ReCAM-based accelerator, which can boost the performance for many typical database operations by flexibly exploiting the inherent parallelism of the ReCAM array. Results showed ReSQM significantly outperform existing CPU, CMOS-based CAM, GPU, FPGA, NDP, and PIM solutions by the orders of magnitude improvement in terms of the speedups of ($2.2\times \sim 39\times$), and ReSQM also achieved $17\times \sim 193\times$ energy saving compared with the CPU baseline.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments and valuable feedback.

References

[1] K. G. Coffman and A. M. Odlyzko, "Internet growth: Is there a "Moore's law" for data traffic?" in *Handbook of Massive Data Sets*. Boston, MA, USA: Springer, 2001, pp. 47–93.

[2] S. Kelling *et al.*, "Data-intensive science: A new paradigm for biodiversity studies," *BioScience*, vol. 59, no. 7, pp. 613–620, 2009.

[3] M. Korkmaz, M. Karsten, K. Salem, and S. Salihoglu, "Workload-aware CPU performance scaling for transactional database systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2018, pp. 291–306.

[4] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *Proc. VLDB*, 1990, pp. 314–325.

[5] G. Smith, *PostgreSQL 9.0 High Performance*. Birmingham, U.K.: Packt Publ., 2010.

[6] D. R. Augustyn and L. Warchal, "GPU-accelerated method of query selectivity estimation for non equi-join conditions based on discrete Fourier transform," in *New Trends in Database and Information Systems II*. Cham, Switzerland: Springer, 2015, pp. 215–227.

[7] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Comput. Surveys*, vol. 24, no. 1, pp. 63–113, 1992.

[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[9] N. Bandi, D. Agrawal, and A. E. Abbadi, "Fast computation of database operations using content-addressable memories," in *Proc. 17th Int. Conf. Database Expert. Syst. Appl. (DEXA)*, 2006, pp. 389–398.

[10] D. Agrawal and A. E. Abbadi, "Hardware acceleration for database systems using content-addressable memories," in *Proc. Int. Workshop Data Manag. New Hardw. (DaMoN)*, 2005, pp. 1–7.

[11] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop Gen. Purpose Comput. Graph. Process. Units (GPGPU)*, 2010, pp. 94–103.

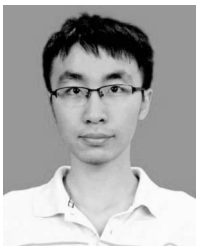
[12] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, 2009, pp. 1–12.

[13] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2010, pp. 347–358.

[14] N. Satish *et al.*, "Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 351–362.

[15] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU join processing revisited," in *Proc. Int. Workshop Data Manag. New Hardw. (DaMoN)*, 2012, pp. 55–62.

[16] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2014, pp. 151–160.
[17] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems," in *Proc. Int. Symp. High Perform. Distrib. Comput. (HPDC)*, 2008, pp. 165–174.
[18] B. Sukhwani *et al.*, "Database analytics acceleration using FPGAs," in *Proc. Int. Conf. Parallel Archit. Comp. Tech. (PACT)*, 2012, pp. 411–420.
[19] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. Dewitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1221–1230.
[20] Y. Kang, Y. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proc. IEEE Symp. Mass Stor. Syst. Tech. (MSST)*, 2013, pp. 1–12.
[21] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul./Aug. 2014.
[22] Y. Sun, Y. Wang, and H. Yang, "Bidirectional database storage and SQL query exploiting RRAM-based process-in-memory structure," *ACM Trans. Stor.*, vol. 14, no. 1, p. 8, 2018.
[23] L. Yavits, A. Morad, and R. Ginosar, "Computer architecture with associative processor replacing last-level cache and SIMD accelerator," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 368–381, Feb. 2015.
[24] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 148–151, Jul.–Dec. 2015.
[25] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A resistive CAM processing-in-storage architecture for DNA sequence alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, Aug. 2017.
[26] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proc. ACM/IEEE Int. Symp. Low Power Elect. Design (ISLPED)*, 2010, pp. 189–194.
[27] S. Blanas and J. M. Patel, "Memory footprint matters: Efficient equi-join algorithms for main memory data processing," in *Proc. Annu. Symp. Cloud Comput. (SOCC)*, 2013, pp. 1–16.
[28] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," in *Proc. VLDB Endow.*, vol. 7, no. 1, 2013, pp. 85–96.
[29] M. Imani, S. Gupta, S. Sharma, and T. S. Rosing, "NVQuery: Efficient query processing in nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 628–639, Apr. 2019.
[30] L. Zhao, Q. Deng, Y. Zhang, and J. Yang, "RFAcc: A 3D ReRAM associative array based random forest accelerator," in *Proc. ACM Int. Conf. Supercomput. (ICS)*, 2019, pp. 473–483.
[31] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proc. Int. Workshop Data Manag. New Hardw. (DaMoN)*, 2015, pp. 1–10.
[32] L. Li, H. Wang, J. Li, and H. Gao, "A survey of uncertain data management," *Front. Comput. Sci.*, vol. 14, no. 1, pp. 162–190, 2020.
[33] M. Zhang, H. Wang, J. Li, and H. Gao, "Diversification on big data in query processing," *Front. Comput. Sci.*, vol. 14, no. 4, 2020, Art. no. 144607.
[34] J. Cao and R. Li, "Fixed-time synchronization of delayed memristor-based recurrent neural networks," *Sci. China Inf. Sci.*, vol. 60, no. 3, 2017, Art. no. 032201.
[35] D. Wang, W. Zhao, W. Chen, H. Xie, and W. Yin, "Fully coupled electrothermal simulation of resistive random access memory (RRAM) array," *Sci. China Inf. Sci.*, vol. 63, no. 8, 2020, Art. no. 189401.



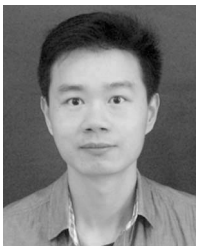
Huize Li (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

His current research interests include computer architecture and emerging nonvolatile memory.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994.

He is a Cheung Kung Scholars Chair Professor of computer science and engineering with the HUST. He worked with the University of Hong Kong, Hong Kong, from 1998 to 2000, and as a Visiting Scholar with the University of Southern California, Los Angeles, CA, USA, from 1999 to 2000. He has coauthored 15 books and published over 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Dr. Jin was awarded Excellent Youth Award from the National Science Foundation of China in 2001. In 1996, he was awarded a German Academic Exchange Service Fellow-Ship to visit the Technical University of Chemnitz in Germany. He is a fellow of CCF and a member of ACM.



Long Zheng (Member, IEEE) received the Ph.D. degree in computer engineering, Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016.

He is currently an Associate Professor with the School of Computer Science and Technology, HUST. His current research interests include program analysis, runtime systems, and configurable computer architecture with a particular focus on graph processing.

Xiaofei Liao (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

He is currently the Vice Dean with the School of Computer Science and Technology, HUST. He has served as a Reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P system, cluster computing, and streaming services.

Dr. Liao is a Member of the IEEE Computer Society.