Relational Memory: Native In-Memory Accesses on Rows and Columns

Shahin Roozkhosh[1‡], Denis Hoornaert[2‡], Ju Hyoung Mun[1], Tarikul Islam Papon[1], Ahmed Sanaullah[3], Ulrich Drepper[3], Renato Mancuso[1], Manos Athanassoulis[1]
[1]Boston University, [2]Technical University of Munich, [3]Red Hat
{shahin,jmun,papon,rmancuso,mathan}@bu.edu,denis.hoornaert@tum.de,{asanaull,drepper}@redhat.com

ABSTRACT

Analytical database systems are typically designed to use a column-first data layout to access only the desired fields. On the other hand, storing data row-first works great for accessing, inserting, or updating entire rows. Transforming rows to columns at runtime is expensive, hence, many analytical systems ingest data in row-first form and transform it in the background to columns to facilitate future analytical queries. *How will this design change if we can always efficiently access only the desired set of columns?*

To address this question, we present a radically new approach to data transformation from rows to columns. We build upon recent advancements in embedded platforms with re-programmable logic to design *native in-memory access on rows and columns.*

Our approach, termed *Relational Memory* (RM), relies on an FPGA-based accelerator that sits between the CPU and main memory and transparently transforms base data to any group of columns with minimal overhead at runtime. This design allows accessing any group of columns as if it already exists in memory. We implement and deploy RM in real hardware, and we show that we can access the desired columns up to 1.63× faster compared to a row-wise layout, while matching the performance of pure columnar access for low projectivity, and outperforming it by up to 2.23× as projectivity (and tuple reconstruction cost) increases. Overall, RM allows the CPU to access the optimal data layout, radically reducing unnecessary data movement without high data transformation costs, thus, simplifying software complexity and physical design, while accelerating query execution.

1 INTRODUCTION

OLTP vs. OLAP vs. HTAP. Over the past few years, large-scale real-time data analytics has soared in popularity as more and more applications need to analyze fresh data. This has been exacerbated by new technological trends like 5G, Internet-of-Things, and the advent of cloud computing as an always-on data platform [24, 34]. This leads to the need for database management systems (DBMS) that can perform both Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP), known as Hybrid Transactional/Analytical Processing (HTAP) [63]. However, OLTP and OLAP systems adopt very different designs. OLTP systems are generally optimized for write-intensive workloads aiming to support high-volume point queries using indexes. In contrast, OLAP systems are optimized for read-only queries that access large amounts of data. Recent efforts for HTAP systems have been bridging OLAP and OLTP requirements by maintaining multiple copies of data in different formats [18, 67] or converting data between different layouts [13, 15, 51, 56, 75].

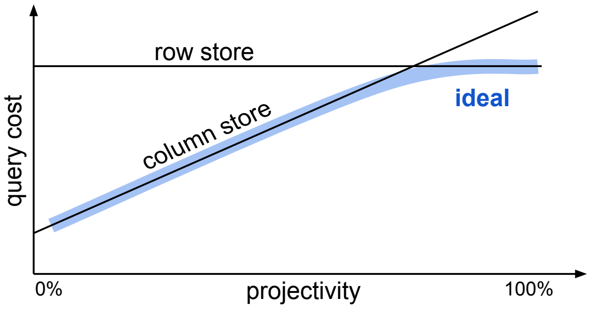


Figure 1: Row-wise accesses have constant cost, while columnar accesses have higher cost for higher projectivity. Ideally, the cost should be the minimum of the two.

Data Layout. Systems typically employ one of the two data layouts: *row-stores* or *column-stores.* **Transactional systems** employ row-stores, meaning that the physical organization of data items in memory is structured in contiguous rows. Row-stores are ideal for queries that append a new row, update the contents of one, or access all attributes. On the other hand, most **analytical systems** store data in a columnar fashion that supports fast scans. Column-stores group together the same attribute of different rows, allowing for efficient analytical query processing [1].

In order to bridge the analytical and transactional requirements, many HTAP systems use a single architecture that ingests data in row-format, and eventually converts them into a columnar format [63]. By doing so, HTAP designs aim to maintain a single data store that can offer data freshness and efficient analytics on that same data set. However, research on adaptive layouts adopted by systems like H_2O [8], Hyper [47], Peloton [15], and OctopusDB [28] shows that every query has an optimal layout that is neither a column-store nor a row-store. Supporting multiple layouts, however, carries a large amount of complexity, which leads to runtime inefficiency arising from heavy book-keeping that also makes the code less scalable and harder to maintain.

What if the optimal layout was always available?

In other words, "*what if the underlying hardware allows us to access only the desired groups of columns while the data is stored in memory as a row-store?*" Such hardware would read only useful data, hence having a query cost tightly correlated with projectivity as shown by the ideal line in Figure 1. It will have to access only useful data without paying a tuple reconstruction cost. Prior work supports adaptive layouts via code generation [8, 46, 47] and fixes the base storage as either a row-store or a column-store, while adapting the layout via copying only the relevant data during query execution. Such an approach creates the need for maintaining coherence between multiple copies upon updates.

Instead, in this paper, we propose a novel hardware design for data reorganization that (1) can be implemented in existing commercial platforms, (2) is capable of on-the-fly interception of CPU-originated memory requests, and (3) of producing responses

[‡]These authors contributed equally.

where the supplied data items are always transparently arranged in the most efficient layout, while (4) the source data tables are always stored in physical memory according to the same format – i.e., as a row-store. We show how to integrate this new hardware design in data systems using clean abstractions, without having to redesign the entire database engine.

Design Goals. The proposed hardware-software co-design approach for data reorganization has the following design goals:

(A) **Reduce data movement** in the memory hierarchy.
(B) Always provide the **optimal layout** for both updates (row-wise) and queries (only the desired columns).
(C) Support **ad-hoc queries** over multiple tables via run-time hardware configuration, and an **intuitive interface**.
(D) Support **transactional semantics** via MVCC.
(E) **Prototype our design as FPGA-based** custom hardware.

1.1 Design Overview

Relational Memory. The proposed specialized hardware acts as an on-the-fly data transformer from rows stored in memory to any group of columns shipped through the memory and cache hierarchy toward the processor. We utilize commercially available systems-on-chip (SoCs) that include programmable logic (PL), typically deployed on field-programmable gate arrays (FPGAs), and a traditional multi-core processing subsystem (PS) on the same chip. These PS-PL SoCs allow the design and deployment of resource management primitives and create *functional proof-of-concept prototypes* to assess practical performance benefits.

Specifically, we capitalize on recent advancements in reprogrammable hardware [69] to implement programmable logic *between the memory and the processor.* To ensure ease of programmability, we do not directly expose the specialized hardware to the data system engineer. Instead, we expose a simple abstraction that allows them to request the desired column groups and transparently use the underlying machinery. We refer to the ability to provide an on-the-fly representation of the data that optimizes relational operators as *Relational Memory.*

Ephemeral Variables. In a DBMS implementation, every relational table loaded in memory is accessible through a variable. By default, this points to the base row-oriented representation of the data, tailored for accessing entire rows and updating or inserting data. To support different layouts over the same base data, we introduce *ephemeral variables*, a special type of variable that identifies a specific subset of columns to access. These variables are never instantiated in main memory. Instead, upon accessing such a variable, the underlying machinery is set in motion and generates an on-the-fly projection of the requested columns according to the format that maximizes data locality.

The philosophy behind Relational Memory pivots on three main points: (1) pushing relational operators closer to data storage; (2) reorganizing and compacting data items before they are moved toward CPUs to improve locality; and (3) relying on traditional CPUs for data processing once good locality has been achieved. Operating closer to the data also introduces opportunities to exploit memory cells' inherent parallelism, e.g., by issuing outstanding parallel requests to separate DRAM banks. Note that hardware prefetching can benefit from the memory cells' parallelism as long as the accesses follow a sequential logic. However, sitting closer to the memory, Relational Memory monitors entire accesses and has semantic knowledge that enables it to perform operations out of sequence and exploit inherent memory parallelism. Reorganizing data to improve locality minimizes the waste of constrained CPU cache real-estate. In turn, this translates to better efficiency for the query at hand and lower cache pollution. Lastly, we enable seamless integration with existing DBMS by limiting our design to data reorganization while relying on CPUs to implement arbitrarily complex analytics.

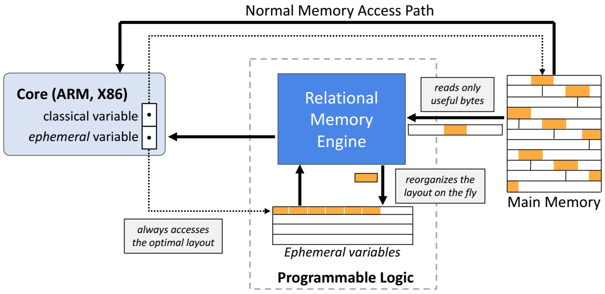


Figure 2: Architecture and data flow of the proposed Relational Memory Engine. The engine pushes projection closer to data and provides the optimal layout via on-the-fly data reorganization.

Figure 2 shows a high-level diagram of the proposed design. The Relational Memory Engine (RME) is located in the programmable logic between the memory and the processor. Once triggered, RME transforms data to any desired column-group allowing the processor to directly access data in the optimal layout through pointers to *ephemeral variables.*

1.2 Contributions

Relational Memory is the first hardware/software co-design that allows near-native access to both rows and column-groups over data stored in a row-wise format in memory, via supporting near-data projection. Native data access to both rows and column-groups leads to better cache utilization and paves the way towards a unified HTAP architecture even in the presence of queries with very different access patterns and requirements. Our work offers the following concrete contributions.

- We present *Relational Memory*, a novel SW/HW co-design paradigm for general-purpose query engines, which ensures that every query has always access to the optimal data layout.
- We propose *ephemeral variables*, a simple and lightweight abstraction to use Relational Memory.
- We implement an FPGA proof-of-concept that demonstrates the viability and impact of our design. The source code is available at https://github.com/ro0zkhosh/relational-memory-engine.
- We tailor a fully configurable design capable of adapting to any query of interest and any data layout at run-time through a user-friendly interface.
- We experimentally show that RME performs native accesses to groups of columns as if the ideal layout is available in memory with no extra cost to transform rows to columns, leading to higher cache efficiency and overall performance.

2 BACKGROUND

We now introduce the necessary background concepts to present the design of Relational Memory. First, we introduce the nuts and bolts of the FPGA technology and the organization of PS-PL platforms. Next, we discuss the Programmable Logic In-the-Middle (PLIM) approach that this work builds upon. Lastly, we discuss the key principles for data layouts.

Field-Programmable Gate Arrays. FPGAs are programmable devices that can be configured to synthesize hardware functional blocks [49, 81]. FPGAs are becoming increasingly popular in

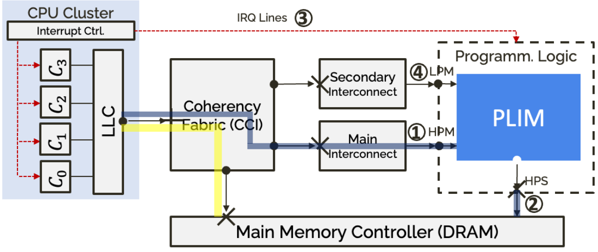


Figure 3: PLIM module instantiation on a PS-PL platform.

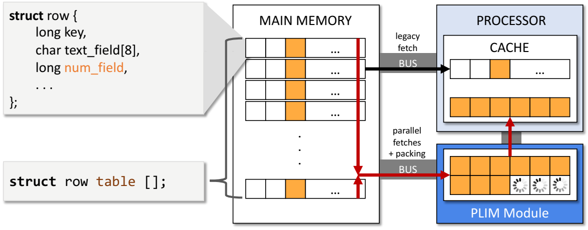
modern platforms because of their high parallelism, reconfigurability, specializability, and power efficiency. In comparison to more traditional CPUs, co-processors, and GPUs, they do not rely on the execution of a set of instructions. Instead, using a *bitstream* mapping of a synthesized version of the logic to its internal components, they are capable of directly emulating the logic of any digital circuit. Because of this unique capability along with specific data manipulation, near-perfect locality of the data, and many levels of parallelism (e.g., pipelining), FPGA technology has been widely used to implement specialized accelerators.

An FPGA device is organized as numerous programmable logic blocks surrounded by an interconnect fabric. *Lookup tables* (LUTs) are the main building block in programmable logic. Each LUT is essentially an n-input, 1-output table to be configured with an arbitrary boolean function [6, 70]. Nowadays, multi-output LUTs are also available [85]. Multiple LUTs can be connected using the configurable interconnect fabric. In addition to the logic circuits, an FPGA has a small memory (registers or flip-flops) and a larger local memory implemented as Block RAM (BRAM), and can access large but slow *off-chip* memories through the dedicated DRAM controllers. Internal memory can reach TB/s scale bandwidth with sub-microsecond latency, whereas off-chip memories' bandwidth can reach GB/s [58].

PS-PL Platforms. Recent years have seen the advent of PS-PL platforms that constitute heterogeneous Systems-on-Chip (SoC) where a traditional processing system (referred to as *PS-side*) is associated with a tightly integrated piece of programmable logic, i.e., an FPGA (referred to as *PL-side*). The adoption of these platforms has gained significant momentum from the recent developments by Intel [40], Xilinx [87], ETHZ [11, 29], and Microsemi with PolarFire SoC [57]. As shown in Figure 3, the PL domain can communicate through high-performance PS-PL interfaces (①, ②, ④), or Interrupt lines (③) with the rest of the system. On-chip communications are carried out using a high-performance, asynchronous, high-frequency, multi-primary/secondary communication interface between functional blocks.

These platforms use the popular, open specification, and widely adopted *Advanced eXtensible Interface* (AXI) protocol [14]. It supports asynchronous **read** and **write** transactions through dedicated channels operating in parallel between a primary (a processor) and secondary (a memory device). In addition, the protocol allows the primaries to emit multiple outstanding transactions. Each sequence of transactions is identifiable via a given ID [14].

Programmable Logic in The Middle (PLIM). Traditionally, the PL-side is used in the PS-PL platform to map hardware accelerators that work in a load-unload fashion [19]. However, the recently introduced PLIM approach [69] creates new design opportunities by using the PL-side as a secondary route to main memory that can entirely or partially replace the *normal* route. As illustrated in Figure 3, instead of using the normal data path (highlighted in yellow), CPU traffic can be redirected through the PL-side before reaching main memory (highlighted in blue).


```
struct row {
      long key,
      char text_field[8],
      long num_field,
      . . .
};
```

```
struct row table [];
```

Figure 4: Example of RME's in-line data transformation.

The principal advantage of PLIM is its ability to intercept, inspect and manipulate any PS-originated memory transactions before they reach the main memory, creating the opportunity to address new open challenges. For instance, PLIM mitigates memory fragmentation caused by address coloring, by simply manipulating each transaction address [69]. Further, PLIM can be integrated into a broader framework in order to address challenges on memory traffic scheduling [38], or on memory-based on-chip denial-of-service attacks from remote cores [39].

Data Layouts. A key decision for any data system is the employed *data layout*, which is tightly connected with the type of workloads it primarily targets. In general, there are two extremes according to which DBMS's store data: the n-ary storage model (*row-stores*) and the decomposition storage model (*column-stores*). Row-stores follow the volcano-style processing model where data is organized as tuples, and all the attributes for each tuple are stored sequentially [37, 64]. This design allows superior performance for OLTP workloads [7] since the queries in transactions generally tend to operate on individual tuples. Traditional DBMS like Oracle [9], IBM DB2 [22], SQL Server [4] follow this paradigm. In contrast, column-stores follow the decomposition storage model. They process data one column at a time, hence are better suited for OLAP workloads [1] where the queries tend to operate on multiple tuples but only access a small fraction of the attributes. Many contemporary data systems like Vertica [52], Actian Vector (formerly Vectorwise [90]), MonetDB [21], Snowflake [26] use columnar storage. Even traditional row-stores have developed new variants that support columnar format [18, 50, 53]. OLTP queries are more efficiently executed in row-stores while OLAP queries are more efficient in column-stores. Systems targeting hybrid workloads support *hybrid layouts* in the form of column-groups via the flexible storage model [15] or adaptive layouts [8, 28, 47]. These systems dynamically adapt the storage layout depending on the workload by keeping the same data in different layouts and by converting the data between row and columnar formats for transactions and analytics, respectively. Because of these conversions and multiple layouts, these systems generally have high complexity, high materialization cost, and heavy book-keeping overheads.

3 RELATIONAL MEMORY

Motivation. We now motivate the proposed *Relational Memory* design. We present a sample analytical query, and we highlight the access patterns we optimize through a low-level example.

One can think of a row in a database table as a struct of the type `struct row` (Figure 4). A row-oriented table is simply an array of rows of the type `struct row []`. If one wants to access only the numeric field (`num_field`) from all the rows, i.e., perform column access, this creates stride-pattern data access, where 8 bytes are accessed every few hundred bytes of data. This is inefficient because (1) each new row always pulls an entire cache line from memory; (2) the large strides are not handled well by hardware prefetchers; and (3) in general, more data is transported from the main memory to the processors than what is strictly required for the requested type of access. In the considered example, with a cache line size of 64 bytes, only 1/8 of a cache line is utilized.

A column-store optimizes these types of accesses by storing each attribute separately, hence allowing for accessing only the numerical field through an array of `num_field` elements of all rows. Indeed, in this case, only data items strictly required for the final computation are transported from main memory, resulting in a highly localized access pattern. However, this comes at the cost of having an inefficient layout for insertion, deletion and increasing tuple reconstruction costs with higher projectivity.

Near-Data Projection. To offer contiguous access to a specific column (or column-group), RME leverages the PLIM paradigm [69], which is conceptually similar to Processing-In-Memory (PIM) [54] and Near-Memory Processing (NMP) [17] as they all execute logic close to memory. The key innovation of RME is that it *creates data that does not exist in main memory*, which *the CPU can use as if it exists* in main memory. As we demonstrate in Section 5, our FPGA prototype provides a significant performance advantage. Note that our long-term vision is to reap additional benefits by embedding RME *within* the memory controller itself.

RME creates memory *aliases* to expose non-contiguous content as if it were contiguous. In other words, RME enables accessing the same content in main memory under different strides, but it can be accessed as if it were stored contiguously from the perspective of the CPU. This is drastically different than traditional scatter-gather strategies [74] initiated by typical DMA-capable accelerators (e.g., SIMD processors). Data transformation in RME is performed in line with the instruction stream via fine-grained information on the exact byte-wise location of data items useful for the computation at hand. More importantly, it allows predicting and exploiting data reuse across processing phases. RME receives as input the intended access stride of the query (that maps the physical addresses of the columns to be accessed) and then issues parallel main memory requests for the target data. Finally, it assembles multiple entries in a single *packed* cache line to be sent to the processor. This abstraction creates *non-materialized aliases of column-groups* which, from the cache perspective, pushes arbitrary subsets of columns in *dense memory addresses* to the memory hierarchy. Hence, RME supports both efficient column- and row-oriented accesses while minimizing CPU cache pollution with unnecessary attributes.

Ephemeral Variables and Programming Model. In order to initialize and deploy the proposed hardware, we propose a lightweight software/hardware interface. Specifically, to use Relational Memory, a query (1) configures RME and (2) points its output stream to the desired variable, termed *ephemeral variable*. Since RME is designed to be generic and support ad-hoc queries over multiple tables on various table geometries[1], it needs to be configurable. This configuration is performed at run-time which can happen very quickly ($\sim 0.3\mu s$). Now, we focus on the semantics of *ephemeral variables* through an example, and we discuss RME configuration in more detail in Section 4.2.

Suppose that a full relational table is loaded in memory and structured as a classic 2-D array, as previously discussed. For instance, consider a relational table that corresponds to the array `struct row table[]`, where each row is defined in Listing 1.

To have direct access to a single column or a group of columns, we create an *ephemeral variable*, and we register with RME. From the CPU's perspective, accessing the newly created ephemeral variable is equivalent to having direct access to a subset of columns with a packed view of the relevant fields.

Listing 1: C-style relational table row definition.

```c
struct row {
    long key;               /* 8 bytes*/
    char text_fld1 [8];     /* 8 bytes */
    char text_fld2 [12];    /* 12 bytes */
    char text_fld3 [20];    /* 20 bytes */
    char text_fld4 [16];    /* 16 bytes */
    long num_fld1;          /* 8 bytes */
    long num_fld2;          /* 8 bytes */
    long num_fld3;          /* 8 bytes */
    long num_fld4;          /* 8 bytes */
    long num_fld5;          /* 8 bytes */
};
```

Following our example, to access only columns `num_fld1`, `num_fld3`, and `num_fld4`, we create an ephemeral variable of the type:

Listing 2: C-style ephemeral type definition.

```c
struct column_group {
    long num_fld1;         /* 8 bytes */
    long num_fld3;         /* 8 bytes */
    long num_fld4;         /* 8 bytes */
};
```

After the first access, any CPU access on ephemeral variables (that leads to a cache miss) is routed to and satisfied by the PL, i.e., it activates the RME. The ephemeral variable with type `column_group` provides access to the three desired columns as a contiguous array. This comes with three advantages. First, only useful information is propagated through the cache hierarchy, dramatically reducing cache pollution and working-set size and thus improving cache reuse and locality. Second, RME orchestrates data access to main memory in a way that is DRAM structure-aware to maximize throughput – much like a DMA would. Third, having turned stride access, potentially spanning multiple pages, into a sequential pattern over a smaller buffer greatly improves the effectiveness of CPU-side prefetching.

The Lifetime of a Memory Access. Here, we demonstrate the lifetime of a memory access targeting an ephemeral variable, through a sample analytic query:

Listing 3: Sample projection+aggregation query.

```sql
SELECT SUM(num_fld1 * num_fld4)
    FROM the_table
    WHERE num_fld3 > 10;
```

This query requires accessing three out of ten columns of the table, and can be evaluated using an ephemeral variable as follows:

Listing 4: Query logic in C language.

```c
struct row the_table[];
char* QUERY = 'SELECT SUM(num_fld1 * num_fld4) FROM the_table WHERE num_fld3 > 10';
/* Autogenerated Code Block - START*/
ephermeral struct column_group {
    long num_fld1;
    long num_fld3;
    long num_fld4;
};
struct column_group* cg;
cg = configure(the_table, QUERY);
/* Autogenerated Code Block - END */
long sum = 0;
for (int i = 0; i < cg.length; i++) {
    if (cg[i].num_fld3 > 10) {
        sum += cg[i].num_fld1 * cg[i].num_fld4;
    }
}
```

Note that this is a simplified code snippet. To optimize for performance, one can implement state-of-the-art approaches like

[1] By *table geometry* we refer to the size and the offset of each row and each column.

predication [10, 16] to avoid branch misprediction and *vectorization* [20] to increase locality and computation efficiency. Orthogonally to those optimizations, we focus on minimizing data movement. Ephemeral variables fetch only relevant columns from memory leading to optimal cache utilization. At the configuration phase, the geometry of the access is defined (i.e., the pattern of scattered accesses on the base data according to the database geometry based on the query – Listing 4, line 10 RHS). Now an ephemeral variable (defined on line 9), can be pointed to the configured RME (line 10 LHS). When the data is first accessed, i.e., when the statement `cg[0].num_fld3 > 10` is evaluated, the RME starts projecting only the relevant columns. We present the design of RME in detail in Section 4.

Hardware-Assisted Data Projection. Current state-of-the-art systems that support hybrid layouts create the desired column groups in software; therefore, the data has to pass through the memory hierarchy and be copied to create the desired layout. In contrast, we propose to make *any layout available on the fly* by creating an **ephemeral variable**. CPU accesses to ephemeral variables are intercepted by our RME that constructs a response to each request by packing only data of interest. Hence, from a CPU's standpoint, the data appears always structured according to the optimal layout for the query. Two key benefits are: (1) we do not duplicate data in memory as the ephemeral variables provide a reorganized view of the original data; and (2) unlike traditional hardware accelerators, the CPU can immediately access partial results without having to wait for the RME to complete a full pass over the original data. In fact, **only** memory requests for non-ready cache-lines are stalled by the RME.

Updates & MVCC Transactions. While Relational Memory offers native access to both rows and columns, the base data are stored in memory in a row-oriented format. We treat all ephemeral variables as *read-only* columns or group-of-columns that accelerate analytical queries. Updates are handled by accessing the *read/write* row-oriented base data. Specifically, new rows are appended to the base data. In order to support updates and deletion, we use two timestamp fields for every row, thus supporting multiple versions. The first timestamp is set when a row is inserted to mark the beginning of its validity. The second timestamp is set upon row deletion or replacement (by a newer version), marking the end of its validity. Every time an ephemeral variable is accessed, it generates the (group of) column(s) that contains the valid rows at the time of the query. Using these timestamps, RME supports multi-version concurrency control (MVCC) transactions through snapshot isolation.

4 H/W DESIGN AND IMPLEMENTATION

To implement RME and offer in-memory data storage in a single format (row-stores) while offering a *re-organized view* of the same data with ideal locality (column-groups), we interpose programmable logic between the CPU and main memory. The data organization in memory never changes, but the semantics of memory accesses performed by the CPUs are redefined on the fly. RME uses knowledge of the relation's geometry to make the accessed data appear as if they are stored in compact projections.

As depicted in Figure 5, RME is comprised of six modules: (1) the *Configuration Port*, (2) the *Monitor Bypass*, (3) the *Trapper*, (4) the *Requestor*, (5) the *Fetch Unit*, and (6) the *Relational Buffers*. RME interacts with the PS through two primary and one secondary AXI ports. This section provides a bird-eye's view of RME's operating mode and the role of its sub-components.

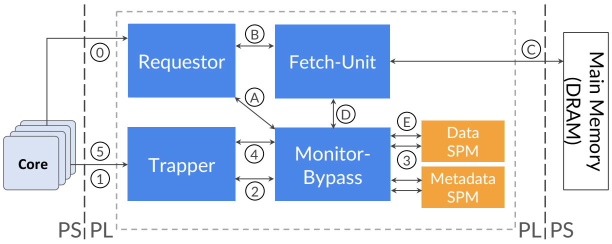


Figure 5: Abstract overview of the RelBuffer components and interconnections with the PS-side.

Configuration Port. The architecture features a configuration port that allows the DBMS to specify the location and geometry (tuple width, tuple count, size, positions of the requested columns) of the target table at runtime. This enables RME to be runtime-configurable and hence to be used for multiple queries. We identify seven parameters as shown in Table 1: (1) *row size* R in bytes; (2) *row count* N; (3) *software reset* SW – that enables the software layer to enforce a reset; (4) *enabled columns* Q, the number of columns of interest (max 11); (5) *column width* C_{A_j}, the width of the j-th column of interest in bytes; (6) *column offset* O_{A_j}, the offset in bytes of the j-th column of interest from the offset of the previous column of interest, $j-1$; and (7) *frame offset* F. Note that, the offset of column j from the beginning of the current row is $\sum_{k=0}^{j} O_{A_k}$. In our prototype implementation, the maximum column width is 64 bytes or one full cache-line and up to 11 non-contiguous columns can be specified. These limitations are not fundamental to our design but only an implementation artifact. We provide an example of how to configure the RME for an arbitrary query on a given database on Section 4.2.

Monitor Bypass. The most central module in this architecture is *Monitor Bypass*, which is in charge of managing, synchronizing, and controlling other modules. It is responsible for (1) interacting with the Trapper to acknowledge and answer incoming requests, (2) collecting data coming from the Fetch Unit, (3) monitoring the completion and availability status of each chunk of reorganized data, and (4) activating the Requestor upon the first CPU-originated request after the RME's configuration.

Trapper. The interfacing between RME's internal logic and the PS-side is performed by the *Trapper*, which is the first module that encounters CPU-originated memory requests targeting the reorganized data. Upon the arrival of a CPU-originated read transaction, the Trapper extracts the target data address and request ID (i.e., the tuple {`A, ID`}) before forwarding them to the Monitor Bypass. This transfer is done through a dedicated unidirectional channel. The Trapper finally formulates AXI-compatible data responses based on the requested reorganized cache-line and the corresponding ID (i.e., {`ID, RD`}) as they are made available by the Monitor Bypass. Since the CPUs can issue multiple asynchronous requests, the Trapper-Monitor Bypass interface has been designed to handle multiple outstanding transactions.

Requestor. The Requestor leverages the data geometry passed via the configuration interface (see Table 1) to orchestrate access to main memory. It generates a deep sequence of *request descriptors* indicating, for each row, the beginning (and length) of valuable data within a set of bus-width-aligned transactions. Each descriptor also indicates the positions where to store the extracted columns. The descriptor is generated such that the resulting main memory requests are always bus-width aligned and with variable burst length, never to fetch more data than needed.

Internally, the Requestor keeps track of the absolute position $P_{i,j}$ at which the useful data starts for the j-th column of the i-th

Parameter	symbol	Address	Description
Row size	R	`base+0x00`	database tuple width
Row count	N	`base+0x04`	database tuple count
Software reset	SW	`base+0x08`	software triggered reset request
# Enabled columns	Q	`base+0x0c`	amount of columns of interest
Column width	C_{A_j}	`base+0x10+(j*0x2)`	j-th column width ($j \in [0, 11]$)
Column offset	O_{A_j}	`base+0x26+(j*0x2)`	j-th column offset ($j \in [0, 11]$)
Frame offset	F	`base+0x3c`	filtered table frame offset

Table 1: RME configuration port: addresses and description.

row, with $i \in [0, N)$ and $j \in [0, Q)$, by computing

$$P_{i,j} = R \cdot i + \sum_{k=0}^{j} O_{A_k}. \tag{1}$$

The i-th descriptor generated by the Requestor is then comprised of five parameters that depend on the platform-specific yet constant bus width B_W. These are (1) the main memory address $R^{\text{addr}}_{i,j}$ to fetch the (i, j)-th chunk of useful data; (2) the burst length $R^{\text{burst}}_{i,j}$ of each main memory request; (3) the position $W^{\text{addr}}_{i,j}$ in the internal RME's buffer where to store the extracted chunk of data; (4) the leading $E^{\text{s}}_{i,j}$ and (5) trailing $E^{\text{e}}_{i,j}$ number of bytes to be discarded in the response received from main memory. These parameters are generated as shown in Eq. (2) through (6), where the "//" operator represents the integer division and the "%" operator represents the remainder of an integer division.

$$R^{\text{addr}}_{i,j} = (P_{i,j}//B_w) \cdot B_w \tag{2}$$

$$R^{\text{burst}}_{i,j} = \lceil((P_{i,j}\%B_w) + C_{A_j})/B_w\rceil \tag{3}$$

$$W^{\text{addr}}_{i,j} = (i-1) \cdot \sum_{k=0}^{Q} C_{A_k} + \sum_{k=0}^{j-1} C_{A_k} \tag{4}$$

$$E^{\text{s}}_{i,j} = P_{i,j}\%B_w \tag{5}$$

$$E^{\text{e}}_{i,j} = (P_{i,j} + C_{A_j})\%B_w \tag{6}$$

Every time a descriptor is produced, it is passed to the Fetch Unit if it is available. When requesting data from main memory, only the locations with relevant data at the granularity of the bus width are accessed. In order to access the desired data efficiently, the Requestor produces descriptors that instruct a given Fetch Unit to perform variable-length memory bursts via the $R^{\text{burst}}_{i,j}$ parameter. This contrasts with a generic cache controller that uses the burst length required to fetch an entire cache-line.

Fetch Unit. The Fetch Unit is responsible for retrieving one fixed-size chunk of data from main memory and then directing it to the designated place in the Reorganization Buffer. The unit is internally structured in several sub-components, namely Reader, Column Extractor, and Writer, which are described below.

The *Reader* directly interacts with the main memory controller. It primarily produces memory requests that reflect the specifications of the descriptor passed by the Requestor. The Reader uses the AXI protocol to perform variable-burst memory requests towards main memory at the granularity of a single bus beat (i.e., bus-width, which is typically a fraction of the cache-line size). To maximize performance, the Reader has been designed to take advantage of *memory-level parallelism*. More specifically, up to 16 simultaneous transactions can be sent to and managed by the DRAM controller. The data payload obtained by the Reader is passed to the Column Extractor module.

The *Column Extractor* module extracts the individual bytes that correspond to the portion of the columns of interest. If the target column(s) spans multiple bus lines; it waits until all the required data items are accumulated and indicates the output's validity by setting the *enable* signal. Once ready, the extracted data is inserted in a buffer called *packer*. The data is appended to the *packer*'s content until an entire cache line (i.e., 64 bytes) is ready. Only then, the cache line is passed onto the Writer module. The *packer* limits the number of accesses to the Reorganization Buffer, reducing the Reorganization Buffer access overhead.

Any write operation emitted by the *Writer* module passes through the Monitor Bypass. The Writer receives the data from the Column Extractor, along with the location where to store the packed data in the Reorganization Buffer. Finally, it forms a single write request to the Reorganization Buffer.

Reorganization Buffer. Two internal memory blocks serve as Scratch Pad Memories (SPMs) to store the data and the metadata. The **Data SPM** stores extracted chunks of data arriving from the Fetch Unit and the **Metadata SPM** stores the bookkeeping information maintained by the Monitor Bypass. For each cache-line, the latter stores the tuple $\{P$, `ID`$\}$, where (1) P is the epoch to which the line belongs to; and (2) `ID` represents the stalled transaction ID. `ID` is non-null only if a transaction requesting the line is *pending*. Upon cache-line delivery from the Fetch Unit, P is updated with the current RME epoch. Next, the Monitor Bypass interprets a status line as *complete* iff P matches the RME's current epoch. The epoch mechanism enables quick invalidation of the content of both SPMs because changing the RME's epoch makes every SPM entry incomplete. The software-triggered reset relies on this mechanism to invalidate the SPMs in a single clock cycle.

4.1 Data-path and Work-flow

We now go over the workflow of RME following a CPU-originated read transaction. We consider two scenarios. (1) First, the case where the requested data has already been fetched from main memory and reorganized. Thus, it can be immediately sent to the requesting CPU (Reorganization Buffer hit). (2) Second, the case where the target data needs to be fetched from main memory (Reorganization Buffer miss). We consider the two cases separately and refer to Figure 5 as we discuss each step of the flow.

RME is designed to be generic, supporting various data layouts. Therefore, the software must initially configure RME with the geometry of the target relation as described in §4 and depicted in Figure 5 ⓪. Once a core emits an AXI read request, it is intercepted by the RME. Next, ① the Trapper extracts the {`A`, `ID`} fields and ② pass them to the Monitor Bypass. The latter checks whether the new request can be immediately served (hit) or if it must be stalled (miss). The check is done using the `A` field to ③ fetch the cache-line status from the **Metadata SPM**. Speculatively, ③ the (possibly valid) content of the requested cache-line is fetched from the **Data SPM**.

Reorganization Buffer Hit. If the cache-line was marked as complete, the Monitor Bypass sends its content – i.e., the tuple {`ID`, `RD`} – to the Trapper ④. Then, the Trapper forms an AXI compliant transaction to reply to the CPU's initial request ⑤.

Reorganization Buffer Miss. If part of the data composing the requested cache-line is missing, the request must be stalled. In this case, the request `ID` is stored in the metadata SPM. Once enough data returns from the Fetch Unit and the cache-line is complete, the `ID` is removed and the {`ID`, `RD`} tuple is sent to the Trapper ④. If the miss in question is the first miss of the frame, a signal is sent to the Requestor to start the descriptor generation. As discussed earlier, the Requestor has a crucial role in orchestrating the Fetch Unit and its interaction with main memory, data extraction, and data forwarding to the Reorganization Buffer. It prepares a series of descriptors for the Fetch Unit using Eq. 1-6.

Upon receipt of a new descriptor, the Fetch Unit Ⓒ sends a request for a burst of $R^{\text{burst}}_{i,j}$ data responses towards main memory at location $R^{\text{addr}}_{i,j}$. Once the full response is received, the Column Extractor performs data filtering using the parameters $E^{\text{s}}_{i,j}$ and $E^{\text{e}}_{i,j}$. Next, the filtered data is inserted in the *Packer* before being sent to the Reorganization Buffer using an address derived from $W^{\text{addr}}_{i,j}$. On its way to the Reorganization Buffer, the filtered data chunks go through the Monitor Bypass Ⓓ. The latter simultaneously updates the record of the newly filled cache-lines in the Metadata SPM Ⓔ. By simultaneously fetching both metadata and data records of the most recently updated cache-line ③, the Monitor Bypass checks whether the cache-line is full. In such a case, the Monitor Bypass immediately sends the corresponding `RD` back to the Trapper ④. The Trapper then replies to the CPU by forming an AXI response using {`ID`, `RD`} ⑤.

4.2 Configuring RME

We now highlight the generality and configurability of RME and provide an example of how a user can configure the engine through the exposed interface. RME is programmed on the PL-side by providing a bitstream *only once at system-boot time*. Subsequently, the DBMS configures RME by writing the necessary parameters in the configuration port. Note that by *(re)configuring*, we refer to the action of writing the memory-mapped configuration registers listed in Table 1 via the software layer. The (re)configuration on average takes less than 0.3 µs. This reconfiguration occurs at the runtime and upon any change to the database layout or the query characteristics.

Suppose a given database table with N rows, each of size R bytes and its first row's address being F. Consider a *Query* that is only interested in Q columns, each C_{A_j}-bytes wide and at offset O_{A_j}. These parameters are passed through the Configuration Port at their corresponding addresses provided in Table 1. Right after the execution of the specific *Query*, a different *Query′* may be submitted, focusing on a different subset of Q' columns, C'_{A_j}-bytes wide at offset O'_{A_j} on the same or a different relational table stored at address F' with N' rows, of size R'. To start processing *Query′* we only need to update these parameters on the Configuration Port after triggering the software reset SW; hence, just a few write operations are enough to reconfigure RME. Note that in our current prototype, the above process of resetting and reconfiguring RME is also used to read from tables the size of which is larger than the available buffer (2MB). In that case, when the buffer is full we trigger a 1-clock-cycle reset to configure RME to fetch data from sequential chunks of the base table by updating only the Frame Address attribute ($F_1, F_2, \ldots$) and triggering the reset SW. The same reconfiguration approach is also adopted when multiple tables are involved in query processing.

5 EVALUATION

We experimentally demonstrate that RME offers efficient native accesses to any column or column-group, outperforming direct row-wise and direct columnar accesses.

5.1 Target Platform

We carry out a full-stack implementation of the proposed RME. Prior to deploying and running our design on real hardware, we extensively validate each of the described sub-modules via simulation-based testing (using Xilinx Vivado). The synthesized hardware is deployed on a real platform which is used to derive all the results presented in this section. We use a Xilinx Zynq UltraScale+ MPSoC platform [87] (ZCU102) equipped with 4 Cortex-A53 1.5 GHz cores, each with a private 32+32 KB L1 I+D cache and sharing a unified 1 MB L2 cache on the PS-side. All our experiments are performed on Linux 4.14, and all the code is compiled using GCC 7.3.1 for AArch64. The final synthesis of RME is integrated on the PL-side. Note that a large chunk of 2 MB **Data SPM** of RME directly affects the critical path and prevents the design from reaching higher frequencies. Thus, the presented RME design (and thus, the PL-side) is constrained to 100 MHz (i.e., one-third of the maximum reachable frequency). This is purely an implementation artifact that can be circumvented at the cost of a few extra clock cycles by splitting the buffer in small memory chunks connected to the Monitor-Bypass via an interconnect-like IP. Note that, despite the constrained synthesis frequency, our prototype already outperforms the pure row-wise and columnar layouts as discussed in Section 5.3.

5.2 Experimental Methodology

Relational Memory Benchmark. We design a synthetic benchmark to test the behavior of RME under a number of representative query access patterns (both single- and multi-tables queries). The benchmark consists of six template queries shown in Listing 5, focusing on projection, selection, aggregation, and join. The queries are executed assuming that all data is in main memory. $Q0$ is the simplest one that calculates an aggregate of a single column. $Q1$ is a projection of k columns (non-contiguous or contiguous), where k can be varied. $Q2$ projects one column and imposes a selection condition on a second column. $Q3$ performs an aggregation (sum) over a subset of a column based on a different selection predicate. $Q4$ further generalizes $Q3$ by adding a `Group By` statement over a third column. Finally, $Q5$ performs a join query over two tables.

Listing 5: Queries 0-5

```
Q0: SELECT SUM(A1) FROM S;
Q1: SELECT A1, A2, ..., Ak FROM S;
Q2: SELECT A1 FROM S WHERE A3 > k;
Q3: SELECT SUM(A2) FROM S WHERE A4 < k;
Q4: SELECT AVG(A1) FROM S WHERE A3 < k GROUP BY A2;
Q5: SELECT S.A1, R.A3 FROM S JOIN R ON S.A2 = R.A2;
```

Implementation. We custom implement an in-memory row-store following the Volcano-style processing model (tuple-at-a-time) and an in-memory column-store following the column-at-at-time processing model. To perform an apples-to-apples comparison, we implement RME, the row-store (ROW) and the column-store (COL) approach in the same codebase. The data types used are `char` (1 byte), `short` (2 bytes), `int` (4 bytes), `long int` (8 bytes), and `__int128_t` (16 bytes).

Experimental Setup. Unless otherwise stated, the default size of each row is 64 bytes and the column-width is 4 bytes. Throughout our experimentation we vary both these parameters to quantify their impact on RME. A column-width smaller than four bytes (one or two bytes) is used to showcase the impact of compression via dictionary encoding. Further, the data size is by default 32 MB and we increase it up to 700 MB to study the ability to handle arbitrary table sizes. When reporting latency, we avoid measurement anomalies by repeating each experiment 30 times and reporting averages and standard deviations. We run two sets of experiments for RME: *hot* (when the targeted data is ready in the Reorganization Buffer) and *cold* (when the targeted data is not yet in the Reorganization Buffer). Although our target FPGA allows RME to have only 2 MB buffer, we believe that *hot* execution cases can provide an overview when RME is equipped with a larger buffer. Note that RME already outperforms traditional layouts in *cold* cases (discussed in Section 5.3), hence local

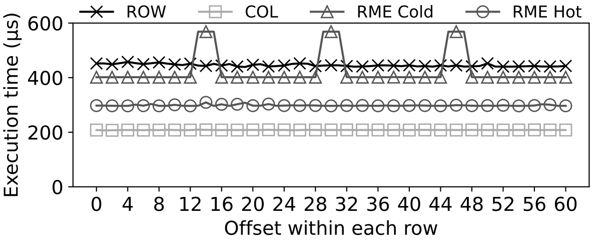


Figure 6: RME outperforms direct accesses to row-oriented memory for Q0. The projected column's offset does not impact RME's performance.

buffering is not fundamental to the performance achieved. We report both *cold* and *hot* execution time for completeness.

5.3 Experimental Results

Column Offset does not Impact Performance. Our first experiment shows the impact of column offset on RME, on an in-memory row-store and on an in-memory column-store. Figure 6 shows the execution time of $Q0$ where we calculate the sum over one column. Table S has 64-byte rows and the width of A_1 is 4 bytes. In the x-axis, we vary the offset of the projected column, and the four different lines correspond to the RME *hot*, *cold*, direct row-wise (ROW) access and direct columnar (COL) access. Figure 6 shows that the projected column A_1's offset O_{A_1} generally does not affect performance, especially for the in-memory row and column store and for *hot* RME. We further observe three spikes (at 13 to 15, 29 to 31, and 45 to 47) for the *cold* cases. This is attributed to the fact that, while most of the time, the 4 bytes of interest would fit within a bus-width (16 bytes) – leading the Requestor to create a read transaction with a burst length of 1 – when the offset plus the data size does not fit in a single bus width (e.g., 13+4, 29+4, 45+4, ...), the Requestor emits read requests with a burst length of 2, increasing access latency. Since the offset of the target column has minimal impact, in the remainder of our experiments, we use column offset 0.

We further note from this figure that even without having the projected column in the Reorganization Buffer (RME *cold*), RME is 14% *faster* than a direct in-memory row-store access (ROW), while RME *hot* access is 50% *faster* than ROW. The reason is that (1) RME better exploits the internal memory bandwidth to fetch only the desired data items at bus-width granularity, and (2) the CPU caches are not polluted with unwanted fields. For this single-column experiment, the latency for COL is, as expected, lower than both ROW and RME. The reason is that Q0 has no materialization (tuple reconstruction) cost. Next experiments show that this behavior changes for higher projectivity.

RME Enables Native Columnar Accesses. Our second experiment shows that RME can efficiently access and propagate through the cache hierarchy individual columns when reading row-oriented data. We now evaluate $Q1$ with $k = 3$ where the three target columns are not contiguous, with offsets $O_{A_1} = 0$, $O_{A_2} = 24$, and $O_{A_3} = 24$ (i.e., A_3 has offset 0+24+24=48 from the beginning of the row) respectively. All three columns have the same width, which is varied between 1 and 16 bytes. Figure 7 shows the normalized execution time of $Q1$. We compare the time to access the data directly from the in-memory row-store, through RME (both *hot* and *cold* accesses), and against in-memory columnar format.

We observe that RME outperforms ROW irrespectively of whether accesses are *cold* or *hot*. The takeaway is threefold. First, *accessing a group of columns via RME* delivers the data with *the optimal layout and outperforms ROW.* Second, *RME achieves an average latency that is comparable to pure columnar accesses.* Specifically, Figure 7 shows that for column size 16 bytes, $Q1$ is faster through RME rather than through a pure column-store. Therefore, data can be simply stored row-wise in memory while any hybrid layout can be delivered by the RME with (almost) no row-to-column data transformation cost. Third, *the performance of RME remains virtually unchanged irrespective of whether the data is ready in the Reorganization Buffer and RME can actually achieve almost the same degree of benefit for the cold cases.* Hence, for the remainder of the text we primarily focus on the cold cases while performing some experiments for both hot and cold accesses to showcase some subtle performance differences.

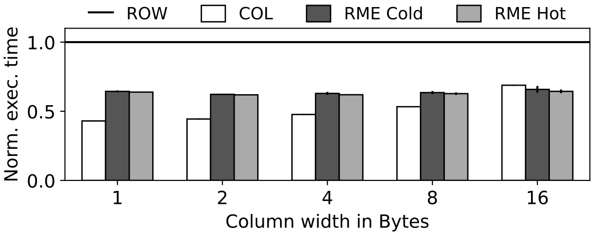


Figure 7: Normalized execution time for Q1 (3 columns). RME outperforms row-oriented direct memory accesses, making our prototype an accelerator that can offer the optimal data layout at a lower latency than DRAM.

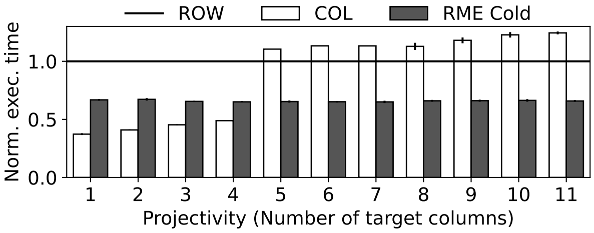
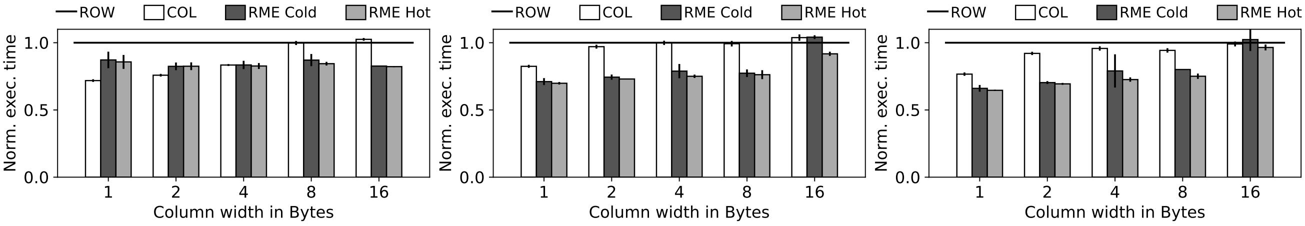


Figure 8: RME has stable performance irrespectively of projectivity when compared with row-wise memory accesses. Further, RME outperforms columnar accesses when projecting more than 4 columns in $Q1$.

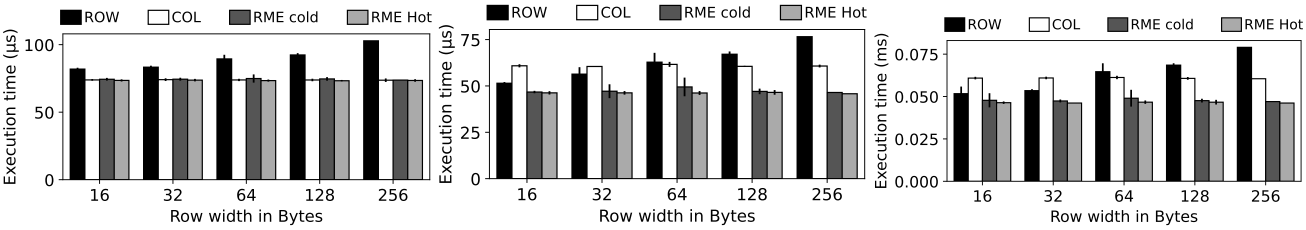
RME Supports Compression via Dictionary Encoding. Figure 7 also shows that RME benefits remain approximately the same as we use fewer bytes to represent the data. Note that in this experiment we have the same domain and we use fewer bytes to represent it exploiting dictionary encoding. We observe that irrespectively of the degree of compression, the benefit of RME over the ROW and COL baselines (which employ the same dictionary encoding) remains the same.

RME has Stable Performance when varying Projectivity. When comparing with columnar accesses, we also have to take into account the tuple materialization cost. In our next experiment, we vary the projectivity from 1 to 11 columns for 4 byte wide columns. Figure 8 shows that for low projectivity (between 1 and 4) reading from a columnar data layout is faster than RME. For projectivity of more than 4 columns, RME outperforms direct columnar accesses because of the tuple reconstruction cost. In addition, through our profiling, we observe that the prefetcher can recognize up to four parallel sequential streams of accesses, which helps the columnar accesses for low projectivity. Overall, RME consistently outperforms direct row-wise accesses that pollute the caches with unwanted fields and outperform columnar access beyond a projectivity threshold.



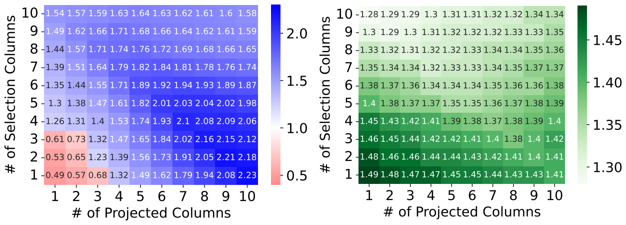
(a) Query 2 (b) Query 3 (c) Query 4

Figure 9: Aggregation queries with varying column size. Depending on the query, the benefit of using RME varies. However, RME outperforms row-oriented direct memory accesses since it accesses only useful data.



(a) Query 2 (b) Query 3 (c) Query 4

Figure 10: Aggregation queries with varying row size. Depending on the query characteristics and projectivity, the benefit varies. However, RME enables more efficient accesses by providing the optimal layout.



(a) Speedup - RME vs Columnar (b) Speedup - RME vs Row

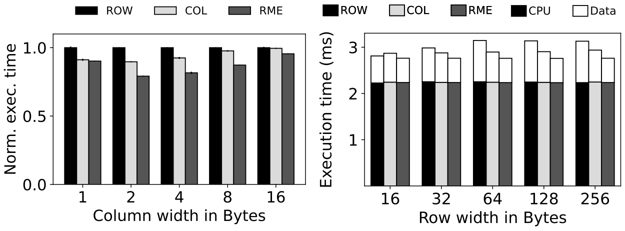
Figure 11: (a) Columnar access performs well when the total number of columns is small (≤ 4). RME dominates when the total number of columns goes beyond 4. (b) RME always outperforms in-memory row access.

Setup for *Q*2 through *Q*5. We continue our experimentation with *Q*2 through *Q*5 on our benchmark from Section 5.2, focusing on the comparison between RME, direct row-wise, and direct columnar access. Two sets of experiments are conducted for each query: (i) we vary the column size of a table with 64 byte-wide rows and (ii) we access 4 byte-wide columns while varying the row size. The default cardinality of the base table is 44K rows.

RME Offers Efficient Near-Memory Projection. We now discuss the performance of *Q*2 that has around 90% selectivity. *Q*2 benefits by fetching only the two desired columns instead of the entire row, while the selection of *Q*2 takes place on the software side. The performance graph is shown in Figure 9a. We observe that RME offers faster execution in both *cold* and *hot* cases. Note that RME particularly outperforms columnar access as the column width increases. Figure 10a shows that the performance gain of RME increases for larger row size (up to 1.4×). We note that *RME's latency remains virtually the same* as it accesses only the relevant data. However, answering the query via direct access of the row-oriented data leads to poor cache utilization as larger rows lead to higher cache pollution. Conversely, RME exhibits stable and predictable performance regardless of the row size.

Selection, Projection, and Aggregation Queries. We now consider the more complex queries *Q*3 and *Q*4, that test selection, projection, aggregation, and group by. The selectivity of *Q*3 and *Q*4 is less than 10%. Similarly to before, we stress the RME using two different sets of experiments. Figure 9b (10b) show the normalized (absolute) latency of *Q*3 when varying the column size with fixed row size (varying the row size with fixed column size). *RME has faster execution* than both ROW and COL in all experiments for *Q*3. For *Q*4, the Group By cost dominates the execution time. Therefore, the performance improvement of RME is reduced, as shown in Figures 9c and 10c. However, RME dominates over direct row-wise access and direct columnar access for both *Q*3 and *Q*4 (complex query with multiple operators) in terms of execution time. We note that both *Q*3 and *Q*4 have a performance drop with 16-bytes columns. This is because in some cases we need to fetch data spanning 32-bytes, i.e., half the cache line size. This happens more frequently due to the low selectivity. As a result, the 2× increase in efficiency in the cache utilization is offset by the overhead of routing through PL memory.

RME Enables Optimal Projection-Selection Queries. In this experiment, we compare RME's performance with the in-memory row-store and in-memory column-store while varying the number of columns for projection and selection in a query. Figures 11a and 11b show the speedup of RME compared to the in-memory columnar access and in-memory row-oriented access. In the x- and y-axis we vary the number of projected columns and the number of columns used for selection from 1 to 10 where the base table has 16 columns (4-byte wide). For this experiment, if the total number of columns used by projection and selection is below 11, distinct columns are selected. Beyond that, some columns might be picked more than once due to our current design constraint of accessing up to 11 columns. Figure 11a shows that when the total number of columns for projection and selection is small (≤ 4), column-store dominates over RME (lower left corner of the heatmap – colored red). However, as the number of column increases, due to the tuple materialization cost and the diminished prefetching benefits, columnar access performance falls behind RME. In fact, RME can be up to 2.23× faster than columnar access (bottom rightmost cell). On the other hand, Figure 11b highlights that RME always outperforms in-memory row



(a) Varying column size

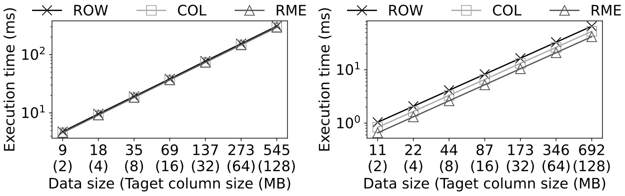
(b) Varying row size

Figure 12: RME performs join faster than traditional row-store join by minimizing data movement.

access by being 1.3 – 1.5× faster. To summarize, RME outperforms the row-store layout because by definition it accesses less data. On the other hand, queries that access fewer columns can be more efficiently evaluated from a columnar layout, however, when the number of projected columns is high enough (more than four in our setup), RME outperforms the columnar layout as well. And it does so in spite of the relatively low synthesis frequency (100 MHz). Operating at a higher frequency may further reduce memory access time and increase the benefits of RME.

RME Reduces Data Movement for Joins. When considering queries that join multiple tables ($Q5$), RME helps to project only the relevant columns, that is, the columns of the join attributes and the column(s) projected in the SELECT statement of the query. In this experiment, we join using a state-of-the-art hash-based join algorithm with a single-pass hash table generation, which is then probed by the second relation. Half of the entries of the outer relation have a match in the inner relation. Figure 12a shows the normalized query latency while varying target column sizes. We observe that joining through RME gives a benefit between 5% and 10% compared to row-wise access. RME outperforms the columnar join as well, providing up to 10% improvement. Figure 12b compares the execution time of this query for different row sizes. RME reduces the total runtime by up to 12% depending on the row width. The graph also shows that the CPU overhead (solid portion of the bars) of hashing constitutes the majority of the runtime which is constant across RME, direct row and columnar access, while RME can optimize the data movement by up to 41% as the row size increases because of its lower cache misses, better-strided accesses, and higher cache utilization.

RME Scales with Data Size. RME supports arbitrary data sizes despite having a small data SPM due to the space limitations imposed by the platform. To evaluate RME's scalability, we use the widely adopted TPC-H [80] benchmark. The current RME implementation supports columns of arbitrary size but not variable length. So, the variable length columns of the TPC-H benchmark are converted to fixed size via padding. To test RME's scalability, we execute TPC-H $Q1$ and $Q6$ in larger tables ranging from 11 MB to 692 MB. Here, the data size for each query is chosen based on the size of target columns. Since $Q1$ uses more attributes than $Q6$, the data size for $Q1$ is smaller than the one for $Q6$. When the size of target columns is larger than the data SPM, RME refills the data SPM. Every time we fill the data SPM, we use the lightweight reset mechanism introduced in Section 4. Figure 13 compares running $Q1$ and $Q6$ using RME vs. direct columnar and direct row access while varying data sizes. Here, the size of target columns is shown in parenthesis. Since $Q1$ consists of group by and sorting, the execution time is similar for all layouts as shown in Figure 13a because the CPU overhead of the query dominates the data movement cost. On the other hand, Figure 13b shows that $Q6$ benefits regardless of the data size, since data movement is the bottleneck, and RME offers the optimal layout.



(a) $Q1$

(b) $Q6$

Figure 13: RME scales with data size and shows stable performance in practical queries such as TPC-H $Q1$ and $Q6$.

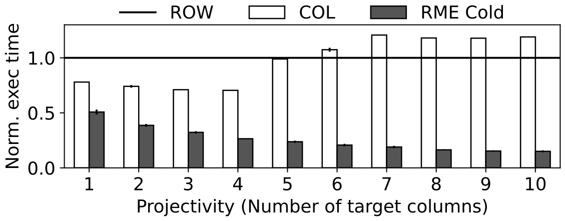
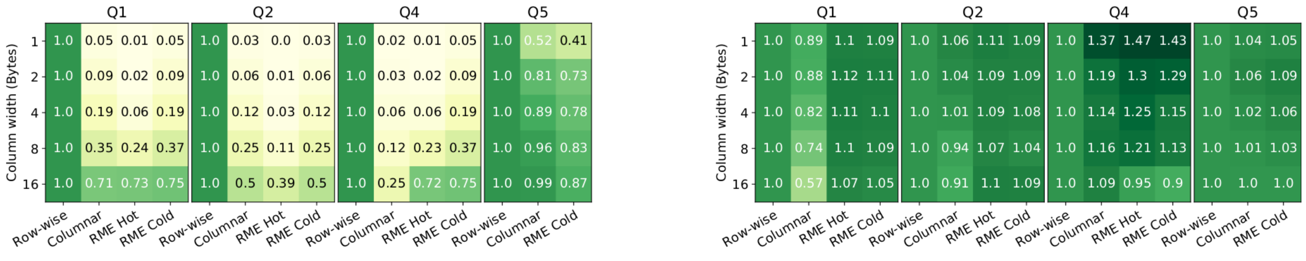


Figure 14: RME outperforms row-wise and columnar accesses as the projectivity grows when MVCC is employed.

RME Supports Lightweight MVCC. RME supports concurrency control by using two timestamps that indicate the beginning and the end of the validity of each row. RME treats the timestamps as additional columns to fetch and check the validity of data in software. We re-run the projectivity experiment with MVCC to evaluate the impact of concurrency control, and show the result in Figure 14. RME supports fetching (up to) 11 distinct groups of columns as shown in Table 1. Since we treat the timestamps as target columns, the maximum number of target columns decreases by one. Note the timestamps can be brought together since they are contiguous. Similarly, it is possible to use all 11 groups of target columns when the target column lies right next to the timestamps. As shown in Figure 14, RME exhibits even better performance compared to Figure 8 as RME gains more in higher projectivity.

Cache Performance and IPC. The benefits from RME observed so far can be further explained if we take a careful look at two key microarchitectural metrics: number of L2 cache misses (refills) and Instructions Per Cycle (IPC) (normalized to the row-wise baseline). Figure 15 shows the two metrics for queries $Q1$, $Q2$, $Q4$, and $Q5$. First, we focus on the L2 refills shown in Figure 15a. We observe that RME enables significantly better cache L2 utilization compared to direct row accesses. Since $Q5$ is more CPU-intensive, the benefit of RME is less prominent whereas for the other queries, the L2 utilization can be as high as 100×. When RME is *hot*, it also dominates direct columnar accesses especially for smaller column widths. When RME is *cold*, we observe an increase in the number of L2 refills, likely due to the prefetching stream experiencing slightly higher memory latency. Nonetheless, the overall effects of the sequential access stride provided by RME are clearly reflected in the measured IPC (Figure 15b). Here, RME in general dominates both row-wise and columnar accesses. Despite efficiently reducing L2 refills, the IPC improvement offered by RME is contained for $Q5$ because of its CPU-bound nature.

RME Supports HTAP Workloads Efficiently. Our experimental analysis shows that RME supports HTAP workloads efficiently without requiring multiple copies of the data. Analytical queries



(a) L2 cache refills

(b) Instructions per Clock Cycles (IPC)

Figure 15: Performance counter measurements for all queries and target memories (Normalized over Row-wise)

Area Report

Resources	LUT	FF	BRAM	DSP
Utilization (%)	2.78	0.68	60.69	0.08

Table 2: Post-implementation area report for the RME design

via RME match or even outperform their column-oriented implementation and transactional accesses are efficiently supported via lightweight hardware-assisted MVCC. Arbitrarily complex relational queries can be implemented from the software side while RME ensures they always use the optimal layout. Overall, our prototype supports HTAP via offering native in-memory columnar accesses over data that is stored in row-oriented format.

5.4 PL Resource Utilization

After the synthesis and the implementation of the design on the ZCU102 development board using Vivado 2017.4, we obtained reports regarding the PL resources utilization of the RME design. As shown in Table 2, the area utilization never exceeds 3% except for BRAM, for which we purposefully maximize the size of the SPMs to improve the performance of RME. The compactness of the design paves the way for more ambitious revisions (see Section 6) and means that the proposed architecture could fit in smaller PS-PL platforms such as the Arty Z7-10 [86], making our approach a good fit for edge and cloud computing as well.

6 DISCUSSION

We now discuss the interaction of RME with various classical database systems decisions and modules, and potential extensions on the hardware front, ranging from memory technology organization to DRAM controller integration.

Data Compression. Relational Memory natively supports dictionary and delta (frame of reference) encoding that are frequently used in state-of-the-art column-store systems [1, 2, 91]. Note that both can be used in row-oriented data, and hence, they can benefit any groups of columns requested by ephemeral variables. Another compression scheme used in column-store systems is a run-length encoding (RLE) [2]. Contrary to dictionary and delta encoding, RLE has an expensive decoding step and relies on data. RLE achieves typically higher compression rates but it is applicable only for sorted data and is more impactful for low-cardinality columns; hence, it is not preferred over dictionary and delta compression [91].

Indexing and Execution Strategies. Typically row-store systems employ indexes, which can be useful when updating the data and for selective queries. In column-store systems (and in some modern row-store systems) [50, 52], column projections are used as a special type of index. RME makes such projections possible without having to materialize them. This has deep implications on the architecture of database systems, since at query time, the query engine may access the data using (i) the base row-oriented version of the data, (ii) an index (if it exists), (iii) the desired columns projected from RME. In this paper, we provide the hardware infrastructure for this and showcase the benefit of accessing any data layout on a per-query basis. One of the opportunities that are enabled through the RME design, is the development of a novel full-fledged hybrid query engine that can alternate between row-at-a-time and column-at-a-time operating over the same base data.

Relational Memory vs. Fractured Mirrors. In this paper, we take a big step from the fractured mirrors approach [66, 67] to offer access to both a row-store and a column-store version of the data, along with everything in-between (arbitrary groups of columns), ***without** maintaining multiple copies of the data.*

Relational Operators in Hardware. The fundamental relational operators are projection, selection, sorting, aggregation, group-by, and join. This paper focuses on projections that require the capability to fetch a subset of the data residing in memory (projecting the desired columns). We implement near-data projection as a proof of concept which lays the groundwork for pushing more processing to hardware. Future work includes implementing selection, aggregates, and join pre-processing in hardware.

RME in Multi-threaded Environment. RME can naturally exploit multi-threaded parallelism as long as all threads access the same data frame. In the current prototype, distinct threads cannot access different RME data frames as a software-triggered reset is needed. This forces the software to proceed frame by frame with appropriate thread-level synchronization. Furthermore, concurrent accesses can be optimized by employing multiple fetch units that work in parallel, an optimization left for future work.

Portability. Our design uses standard Verilog language for the hardware design and C for benchmarks. Hence, the migration to other FPGA chips such as Altera or other processors would not require comprehensive changes. Although the primary target of our prototype is a Xilinx UltraScale+ platform [87], RME can be deployed on many platforms such as Intel's Stratix [40], ETH's Enzian [11, 25, 29], and Microsemi's PolarFire [57] thanks to the generic AXI interfaces it uses. In addition, RME can be deployed on platforms using other protocols for interfacing between the PS and the PL sides. This will require the modification of Trapper and Reader (e.g., changing the AXI interface to PCIe), or to use "bridging IPs" that convert one protocol to another.

Extensions. We now discuss two major future directions.

Towards Higher Memory Controller Utilization. The proposed Requestor module can be extended to interface with multiple Fetch Units in order to decouple the request descriptor generation and the extraction of relevant data. This extension would open the door to performance improvements by exploiting the memory organization. For instance, the DRAM technology is structured around BANKS that can be accessed near-simultaneously, providing higher bandwidth [36, 44]. The extension would require the Fetch Units to indicate their availability and, if all the Fetch Units are busy, stall the Requestor until one to become available. Ideally, the optimal number of Fetch Units should maximize the memory bandwidth utilization while keeping the memory controller below its saturation point.

Memory Controller Integration. Unlike direct access to DRAM, RME forces transactions to cross through a lower-frequency domain, i.e., that of the PL (100 MHz in our case). This clock domain crossing also introduces additional delay for every transaction [38, 39, 69]. Nonetheless, as we observe in our experiments, the benefits unlocked by RME fully offset the described effect. This observation showcases the impact of our long-term vision. On one hand, our RME is expected to provide bigger performance benefits in newer platforms with better PS-PL integration and lower-latency communication interfaces. Moreover, the ability to offer significant performance advantages even at low frequencies makes RME suitable for integration within the main memory controller, pushing the RME functionality inside the controller.

7 RELATED WORK

Hybrid Layouts. Following the *one size does not fit all* rule [78], many HTAP systems use the row-format to ingest data and then convert it to columnar-format for analytical processing [63]. Examples include SAP HANA [32], Oracle TimesTen [51], MemSQL [75], BatchDB [55], and L-store [71]. These HTAP systems fuse the data ingestion and data analytics pipelines. The *optimal* layout is more often neither a column-store or a row-store [8]. Systems like H_2O [8], Hyper [47], Peloton [15], and OctopusDB [28] use adaptive layouts depending on the query patterns. For example, OctopusDB maintains several copies of a database stored in different layouts by maintaining a logical log as its primary storage and then creating secondary physical layouts from this log. H_2O dynamically adapts the storage layout depending on the workload by materializing parts of the data based on the query and as the workload changes, the storage and access patterns keep adapting. Peloton also uses an adaptive policy, however instead of an *immediate* policy, it adopts an *incremental* data reorganization policy. BatchDB proposes the logical separation of analytical queries and transactional updates using (i) data replication, (ii) batch scheduling of queries and updates, and (iii) efficient algorithms for updates [55]. On the other hand, L-Store combines the real-time processing of transactional and analytical workloads within a single unified engine by introducing a novel update-friendly lineage-based storage architecture [71]. All these systems need to store multiple layouts of the data and need to convert between formats which increases the complexity, materialization overhead and maintenance cost.

FPGA in DBMS. FPGAs can be integrated either by using it as a filter or as a co-processor to accelerate the workload [31, 41–43]. In the former approach, the FPGA is used as a decompress-filter between the data source and the CPU to improve the effective bandwidth. This approach has been adopted by systems like AxleDB [72], Netezza [33], Mellanox [61], and Napatech [60]. In contrast, in the latter approach, the FPGA can access the host memory directly and communicate with the CPU via shared memory, thus avoiding the extra copying of data from/to the device memory. Systems like DoppioDB [76], Oracle DAX for SPARC [62] have deployed FPGAs as co-processors. Another technique is to integrate FPGA to the CPU as an I/O device especially where CPUs are the bottleneck. Here the CPU and the FPGA have their own memories. Upon requests from the CPU, the FPGA copies the data from the host memory to the device memory, then it processes the data, writes the results back to the device memory after processing, and finally copies the results back to the host memory. Systems like Kickfire's MySQL Analytic Appliance, dbX have implemented this architecture [23, 73].

There have been attempts to accelerate various DBMS operators like selection [79], aggregation [27], compression [65], decompression [30], sort [89], group by [3], and joins [35, 88]. Moreover, there are approaches to off-load the query itself. Examples include Ibex [82], Q100 [83, 84], and FQP [59]. These approaches receive the SQL query as an input, process the query on the hardware, and pass only the result to CPU. These approaches reduce the computation needed from and the data transfer to the CPU, however, they are not generic enough to support ad-hoc queries. Another interesting line of work is query accelerators in the network layer [5, 12, 48, 68, 77] accessing non-local memory. These works aim to reduce the number of network traversals to access non-local memory, data movement inefficiencies, and network overhead by accelerating operators. Particularly, Farview [48] accelerates a wide range of operators such as projection, selection, aggregation, grouping, and encryption/decryption.

Although many FPGA-based accelerators report high throughput, the low bandwidth between FPGA and host memory (or CPU) is a bottleneck [45]. Thus, designing accelerators is challenging for systems with unpredictable memory access patterns.

In contrast to these approaches, we present a new approach, RME, that transparently transforms data from rows to columns with the help of an FPGA-based accelerator. The strength of our proposed RME is that it does not require any specific implementation for each use case. Rather, RME provides the optimal data layout while *any* ad hoc query's logic can be implemented in software. Hence, RME does not require any data duplication which results in good generalization and wide applicability.

8 CONCLUSION AND FUTURE WORK

In this paper, we present *Relational Memory*, a new design that offers efficient access to both row-oriented and columnar layouts. We build on recent developments in reprogrammable hardware to implement logic between the memory and the processor, which is able to on-the-fly convert rows to arbitrary groups of columns. Our approach pushes projection from software to hardware and enables native access to row- and column-oriented data layouts. Our prototype implementation accesses arbitrary groups of columns at no additional latency than accessing directly the optimal data layout.

Overall, pushing projection to hardware via *Relational Memory* opens up possibilities for radical changes in various database systems components including physical design, indexing, query processing, and query optimization. Further, more relational operators can be implemented in hardware as generic combinable building blocks (selection, aggregation, group by, join preprocessing) reducing the CPU burden. Finally, the low-end hardware used in our prototype underlines that it is feasible to integrate RME in memory controllers, widening its impact.

Acknowledgments. This work is partially funded by NSF Grants CCF-2008799, IIS-2144547, and IIS-1850202, by a RedHat Research Incubation Award, and a Cisco gift. Denis Hoornaert is supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

REFERENCES

[1] Daniel J Abadi, Peter A Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[2] Daniel J Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 671–682. https://doi.org/10.1145/1142473.1142548

[3] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J Halstead, Walid A Najjar, and Vassilis J Tsotras. 2016. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 11:1—-11:9. https://doi.org/10.1145/2933349.2933360

[4] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1110–1121.

[5] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. 120–126. https://doi.org/10.1145/3317550.3321433

[6] Elias Ahmed and Jonathan Rose. 2004. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration Systems (IEEE VLSI Systems)* 12, 3 (2004), 288–298. https://doi.org/10.1109/TVLSI.2004.824300

[7] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a modern processor: Where does time go?. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 266–277. http://128.105.2.28/pub/techreports/1999/TR1394.pdf

[8] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1103–1114. https://doi.org/10.1145/2588555.2610502

[9] Nicole Alexander, Xavier Lopez, Siva Ravada, Susie Stephens, and Jack Wang. 2005. RDF Data Model in Oracle. *Oracle White Paper* (2005). http://download.oracle.com/otndocs/tech/semantic_web/pdf/w3d_rdf_data_model.pdf

[10] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 177–189. https://doi.org/10.1145/567067.567085

[11] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2020/papers/p30-alonso-cidr20.pdf

[12] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the EuroSys Conference (EuroSys)*. 14:1—-14:16. https://doi.org/10.1145/3342195.3387522

[13] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf

[14] ARM. 2019. *AMBA AXI and ACE Protocol Specification.* Technical Report. https://developer.arm.com/documentation/ihi0022/h

[15] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 583–598. https://doi.org/10.1145/2882903.2915231

[16] David I August, Wen-mei W Hwu, and Scott A Mahlke. 1997. A Framework for Balancing Control Flow and Predication. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 92–103. https://doi.org/10.1109/MICRO.1997.645801

[17] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014), 36–42. https://doi.org/10.1109/MM.2014.55

[18] Ronald Barber, Guy M. Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. 2015. In-Memory BLU Acceleration in IBM's DB2 and dashDB: Optimized for Modern Workloads and Hardware Architectures. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

[19] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio C Buttazzo. 2016. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. 1–12. https://doi.org/10.1109/RTSS.2016.010

[20] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. https://doi.org/10.1145/1409360.1409380

[21] Peter A. Boncz, Marcin Zukowski, and Niels J. Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[22] Michael Cain and Kent Milligan. 2011. IBM DB2 for i indexing methods and strategies. *IBM White Paper* (2011). https://tinyurl.com/IBM-DB2-indexing-2011

[23] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 151–160. https://doi.org/10.1145/2554688.2554787

[24] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. *White Paper* (2018). https://virtualization.network/Resources/Whitepapers/0b75cf2e-0c53-4891-918e-b542a5d364c5_white-paper-c11-738085.pdf

[25] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 434–451. https://doi.org/10.1145/3503222.3507742

[26] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W Lee, Ashish Motivala, Abdul Q Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 215–226. https://doi.org/10.1145/2882903.2903741

[27] Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2013. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 25–28. https://doi.org/10.1109/FCCM.2013.38

[28] Jens Dittrich and Alekh Jindal. 2011. Towards a One Size Fits All Database Architecture. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. 195–198. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper25.pdf

[29] ETHZ. 2021. Enzian Systems. *http://enzian.systems/* (2021).

[30] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H Peter Hofstee. 2019. A Fine-Grained Parallel Snappy Decompressor for FPGAs Using a Relaxed Execution Model. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 335. https://doi.org/10.1109/FCCM.2019.00076

[31] Jian Fang, Yvo T B Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* 29, 1 (2020), 33–59. https://doi.org/10.1007/s00778-019-00581-w

[32] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin* 35, 1 (2012), 28–33. http://sites.computer.org/debull/A12mar/hana.pdf

[33] Phil Francisco. 2011. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks* (2011). http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf

[34] Gartner. 2017. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://tinyurl.com/Gartner2020.

[35] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper12.pdf

[36] Mohamed Hassan. 2020. Reduced latency DRAM for multi-core safety-critical real-time systems. *Real-Time Systems* 56, 2 (2020), 171–206. https://doi.org/10.1007/s11241-019-09338-8

[37] Joseph M. Hellerstein, Michael Stonebraker, and James R Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (2007), 141–259. https://doi.org/10.1561/1900000002

[38] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. 2021. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 196. 2:1—-2:22. https://doi.org/10.4230/LIPIcs.ECRTS.2021.2

[39] Denis Hoornaert, Shahin Roozkhosh, Renato Mancuso, and Marco Caccamo. 2021. Work in Progress: Identifying Unexpected Inter-core Interference Induced by Shared Cache. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. https://doi.org/10.1109/RTAS52030.2021.00066

[40] Intel, Corp. 2016. Intel's Stratix 10 FPGA: Supporting the Smart and Connected Revolution. https://tinyurl.com/IntelStratix2016 Accessed on 09.01.2020.

[41] Zsolt István. 2019. The Glass Half Full: Using Programmable Hardware Accelerators in Analytics. *IEEE Data Engineering Bulletin* 42, 1 (2019), 49–60. http://sites.computer.org/debull/A19mar/p49.pdf

[42] Zsolt István. 2020. Let's add transactions to FPGA-based key-value stores!. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 13:1—-13:3. https://doi.org/10.1145/3399666.3399909

[43] Zsolt István, Kaan Kara, and David Sidler. 2020. FPGA-Accelerated Analytics: From Single Nodes to Clusters. *Found. Trends Databases* 9, 2 (2020), 101–208. https://doi.org/10.1561/1900000072

[44] Bruce Jacob, W. Spencer Ng, and T. David Wang. 2008. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers Inc.

[45] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based Data Partitioning. In *Proceedings of the ACM SIGMOD International Conference on*

Management of Data. 433–445. https://doi.org/10.1145/3035918.3035946

[46] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1119–1130. https://doi.org/10.14778/2732977.2732986

[47] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[48] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf

[49] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. *FPGA architecture: Survey and challenges.* Now Publishers Inc.

[50] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zait. 2015. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

[51] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Engineering Bulletin* 36, 2 (2013), 6–13. http://sites.computer.org/debull/A13june/TimesTen1.pdf

[52] Andrew Lamb, Matt Fuller, and Ramakrishna Varadarajan. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801. http://dl.acm.org/citation.cfm?id=2367518

[53] Per-Ake Larson, Remus Rusanu, Mayukh Saubhasik, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, and Srikumar Rangarajan. 2013. Enhancements to SQL server column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1159–1168. https://doi.org/10.1145/2463676.2463708

[54] Gabriel Loh, Nuwan Jayasena, Mark Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. 2013. A Processing in Memory Taxonomy and a Case for Studying Fixed-function PIM. In *Proceedings of the Workshop on Near-Data Processing (WoNDP)*. http://www.cs.utah.edu/wondp/wondp2013-paper2-final.pdf

[55] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 37–50. https://doi.org/10.1145/3035918.3035959

[56] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *Proceedings of the Datenbanksysteme fur Business, Technologie und Web (BTW)*. 545–563.

[57] Microsemi — Microchip Technology Inc. 2020. PolarFire SoC - Lowest Power, Multi-Core RISC-V SoC FPGA. https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga Accessed on 09.01.2020.

[58] Seungwon Min, Sitao Huang, Mohamed El-Hadedy, Jinjun Xiong, Deming Chen, and Wen-Mei Hwu. 2019. Analysis and Optimization of I/O Cache Coherency Strategies for SoC-FPGA Device. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. 301–306. https://doi.org/10.1109/FPL.2019.00055

[59] Mohammedreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible Query Processor on FPGAs. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1310–1313. https://doi.org/10.14778/2536274.2536303

[60] Napatech. 2018. Napatech SmartNIC solution for hardware offload. *https://www.napatech.com/support/resources/solution-descriptions/napatech-smartnic-solution-for-hardware-offload/* (2018).

[61] NVDIA. 2020. NVIDIA Mellanox Innova-2 Flex. *https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/* (2020).

[62] Oracle. 2021. DAX. *https://blogs.oracle.com/linux/post/oracle-data-analytics-accelerator-dax-for-sparc* (2021).

[63] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/-Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. https://doi.org/10.1145/3035918.3054784

[64] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. 2001. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 567–574. https://doi.org/10.1109/ICDE.2001.914871

[65] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[66] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2002. A Case for Fractured Mirrors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 430–441. https://doi.org/10.1007/s00778-003-0093-1

[67] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2003. A Case for Fractured Mirrors. *The VLDB Journal* 12, 2 (2003), 89–101. https://doi.org/10.1007/s00778-003-0093-1

[68] Amazon Redshift. 2021. AQUA (Advanced Query Accelerator) for Amazon Redshift. (2021). https://aws.amazon.com/redshift/features/aqua/

[69] Shahin Roozkhosh and Renato Mancuso. 2020. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 296–309. https://doi.org/10.1109/RTAS48715.2020.00006

[70] Jonathan Rose, Abbas El Gamal, and Alberto L Sangiovanni-Vincentelli. 1993. Architecture of field-programmable gate arrays. *Proc. IEEE* 81, 7 (1993), 1013–1029. https://doi.org/10.1109/5.231340

[71] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 540–551. https://doi.org/10.5441/002/edbt.2018.65

[72] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sönmez. 2017. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems* 51 (2017), 142–164. https://doi.org/10.1016/j.micpro.2017.04.018

[73] Todd C Scofield, Jeffrey A Delmerico, Vipin Chaudhary, and Geno Valente. 2010. XtremeData dbX: An FPGA-Based Data Warehouse Appliance. *Comput. Sci. Eng.* 12, 4 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.93

[74] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2015. Gather-scatter DRAM: in-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 267–280. https://doi.org/10.1145/2830772.2830820

[75] Nikita Shamgunov. 2014. The MemSQL In-Memory Database System. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*.

[76] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. 2017. doppioDB: A hardware accelerated database. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. https://doi.org/10.23919/FPL.2017.8056864

[77] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: smart remote memory. In *Proceedings of the EuroSys Conference (EuroSys)*. 29:1—-29:16. https://doi.org/10.1145/3342195.3387519

[78] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2–11. https://doi.org/10.1109/ICDE.2005.1

[79] Xuan Sun, Chun Jason Xue, Jinghuan Yu, Tei-Wei Kuo, and Xue Liu. 2021. Accelerating data filtering for database using FPGA. *Journal of Systems Architecture* 114 (2021), 101908. https://doi.org/10.1016/j.sysarc.2020.101908

[80] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).

[81] Stephen M Trimberger. 2012. *Field-programmable gate array technology.* Springer Science & Business Media.

[82] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974. http://www.vldb.org/pvldb/vol7/p963-woods.pdf

[83] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: the architecture and design of a database processing unit. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 255–268. https://doi.org/10.1145/2541940.2541961

[84] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2015. The Q100 Database Processing Unit. *IEEE Micro* 35, 3 (2015), 34–46. https://doi.org/10.1109/MM.2015.51

[85] Xilinx. 2017. UltraScale Architecture Configurable Logic Block: User Guide. (2017). https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb

[86] XILINX. 2021. Arty Z7-10: APSoC Zynq-7000 Development Board for Makers and Hobbyists. *https://www.xilinx.com/products/boards-and-kits/1-1630gpl.html* (2021).

[87] Xilinx, Inc. 2016. Zynq UltraScale+ MPSoC - All Programmable Heterogeneous MPSoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html Accessed on 09.01.2020.

[88] Meiting Xue, Qianjian Xing, Chen Feng, Feng Yu, and Zhen-Guo Ma. 2020. FPGA-Accelerated Hash Join Operation for Relational Databases. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67-II, 10 (2020), 1919–1923. https://doi.org/10.1109/TCSII.2019.2959661

[89] Chi Zhang, Ren Chen, and Viktor K Prasanna. 2016. High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS Workshops)*. 148–155. https://doi.org/10.1109/IPDPSW.2016.117

[90] Marcin Zukowski and Peter A. Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27. http://sites.computer.org/debull/A12mar/vectorwise.pdf

[91] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 59. https://doi.org/10.1109/ICDE.2006.150