On-the-fly Data Transformation in Action

Ju Hyoung Mun
Konstantinos Karatsenidis
Boston University
jmun@bu.edu
karatse@bu.edu

Tarikul Islam Papon
Shahin Roozkhosh
Boston University
papon@bu.edu
shahin@bu.edu

Denis Hoornaert
Technical University of Munich
denis.hoornaert@tum.de

Ulrich Drepper
Ahmed Sanaullah
Red Hat
asanaull@redhat.com
drepper@redhat.com

Renato Mancuso
Manos Athanassoulis
Boston University
rmancuso@bu.edu
mathan@bu.edu

ABSTRACT

Transactional and analytical database management systems (DBMS) typically employ different data layouts: row-stores for the first and column-stores for the latter. In order to bridge the requirements of the two without maintaining two systems and two (or more) copies of the data, our proposed system *Relational Memory* employs specialized hardware that transforms the base row table into arbitrary column groups at query execution time. This approach maximizes the cache locality and is easy to use via a simple abstraction that allows transparent on-the-fly data transformation. Here, we demonstrate how to deploy and use Relational Memory via four representative scenarios. The demonstration uses the full-stack implementation of Relational Memory on the Xilinx Zynq UltraScale+ MPSoC platform. Conference participants will interact with Relational Memory deployed in the actual platform.

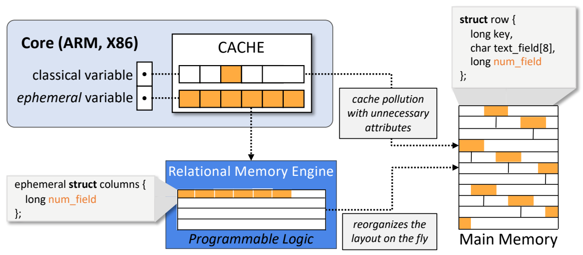
PVLDB Reference Format:
Ju Hyoung Mun, Konstantinos Karatsenidis, Tarikul Islam Papon, Shahin Roozkhosh, Denis Hoornaert, Ulrich Drepper, Ahmed Sanaullah, Renato Mancuso, and Manos Athanassoulis. On-the-fly Data Transformation in Action. PVLDB, 16(12): 3950 - 3953, 2023.
doi:10.14778/3611540.3611593

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BU-DiSC/relational-memory/blob/master/vldb2023.

1 INTRODUCTION

Data Layout: Row-Store vs Column-Store. A major design decision for any data system is whether they follow the row-store or the column-store paradigm. This decision has a profound impact on the entire data system architecture. **Transactional systems** typically employ row-stores, i.e., data blocks are physically organized in memory as contiguous rows. Row-stores provide better performance for transactional workloads (append/update a row, or

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611593



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

```
ephemeral struct columns {
   long num_field
};
```

Figure 1: RME pushes projection closer to data to provide the optimal layout via on-the-fly data transformation.

access all attributes). In contrast, most **analytical systems** store data in a columnar fashion. Since column-stores group together the same attribute of different rows, they allow fast scans and efficient analytical query processing [1]. The several-decade-long journey of these two systems has led to a new family of **hybrid transactional/analytical processing (HTAP)** architectures [4]. Recent efforts for HTAP systems attempt to bridge the transactional and analytical requirements by proposing systems that maintain multiple copies of data in different physical layouts and convert them into the desired layout as required [2, 3]. Because of data duplication, the additional bookkeeping, and the cost of converting data across different layouts, these systems compromise between efficient analytics and data freshness, which leads to runtime inefficiency, less scalability, and poor maintainability.

Hardware Specialization can Help. The idea of *hardware specialization*, although recurring every few years, has not been able to achieve its true potential because of the historically exponential growth of processor speed. However, with the tapering of Moore's law and the exponential growth of data processing needs along with the advancements in reconfigurable logic, hardware specialization is now becoming a more feasible and scalable alternative to general-purpose computing [7]. We ask the question:

Can we access any arbitrary data layout using near-data processing via specialized hardware?

In other words, "*Can we access the optimal data layout using hardware specialization?*". This removes the need to maintain multiple layouts and the overheads associated with it. Further, we can perform efficient analytics over the fresh data without any duplication or conversion. Thus, such a specialized hardware can blend the benefits of both row-stores and column-stores by accessing only the relevant data (without accessing unnecessary data and without paying a tuple reconstruction cost) while maintaining a single layout, consequently leading to better cache utilization.

Our Approach. The research that led to this demo paper proposes a novel hardware design for *on-the-fly* data transformation that intercepts CPU-originated memory requests and generates the optimal layout, while the source data are always stored as a row store in physical memory. We refer to the ability to provide an on-the-fly representation from rows stored in memory to any group of columns as **Relational Memory** [6]. We utilize commercially available systems-on-chip (SoCs) that include both programmable logic (PL) and a traditional multi-core processing subsystem (PS), where we implement programmable logic between the memory and the processor as **Relational Memory Engine (RME)** (Figure 1). RME exposes a carefully designed API, termed *ephemeral variables* that enable accessing arbitrary column groups using simple abstractions to transparently use the underlying machinery. The API creates non-materialized aliases of column-groups, which supports both efficient column- and row-oriented accesses while minimizing CPU cache pollution without any data duplication.

Demonstration. Conference participants can interact with RME deployed in a commercially available PS-PL platform. Participants will be able to see (i) how to configure RME based on the database geometry (number of rows and columns, column widths, column types, etc.), (ii) how to run sample benchmark queries (aggregation, selection/projection, join over two tables) using RME and *ephemeral variables*, and (iii) how RME compares with classical row-stores and column-stores under different scenarios.

2 RELATIONAL MEMORY

Relational Memory is a novel hardware/software co-design for on-the-fly data transformation [5, 6]. Figure 1 shows a high-level diagram of Relational Memory Engine (RME), which is specialized hardware for on-the-fly data transformation that sits on the programmable logic in between the CPU and main memory. RME provides *effortless locality* for any queries without accessing unnecessary data via a simple abstraction called Ephemeral variable. We now discuss the details of RME and ephemeral variables.

Relational Memory Engine (RME). RME offers contiguous access to a specific set of columns in memory since RME reorganizes data on the fly in a format that maximizes the cache locality. In other words, RME transforms the row-oriented base table into the optimal data layout for any query. Figure 2 presents the birds-eye-view hardware architecture of RME. There are four modules: Trapper, Monitor-Bypass, Requestor, and Fetch-Unit, and two Scratch Pad Memories (SPMs) to buffer reorganized data (Data SPM) and its availability (Metadata SPM). RME needs to know the geometry of DB and the set of columns to be transformed. Thus, configuring RME is the first step before accessing the reorganized data (⓪ in Figure 2). Trapper is the interface between the CPU and RME that intercepts read requests (①) from the CPU. Trapper notifies Monitor-Bypass (②) to check the availability of the requested data (③). When the requested data are already in Data SPM, Monitor-Bypass sends the data to Trapper (④), and then, the CPU receives the requested data via RME (⑤). If the requested data is not in the

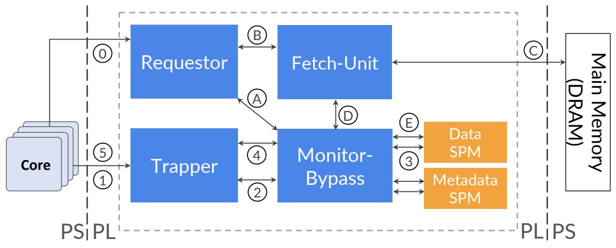


Figure 2: The architecture and workflow of RME.

Data SPM, Monitor-Bypass lets Requestor know about missing data (Ⓐ). Based on the DB geometry, Requestor creates descriptors that identify the location of the desired columns (Ⓑ). Fetch-Unit reads the bus line that contains useful data and extracts the relevant part only (Ⓒ) and sends the extracted part to Monitor-Bypass (Ⓓ) so that it can be stored in Data SPM (Ⓔ). Thus, RME transforms data in a format that minimizes cache pollution.

Ephemeral Variables. Ephemeral variable is a lightweight abstraction to access the reconstructed tuple by RME. Ephemeral variable creates memory *alias* that is never instantiated in main memory; however, it acts like a regular variable from the CPU's perspective, so the CPU can use as if *the data already exist in main memory.* Upon accessing such a variable, the underlying RME is set in motion and generates an on-the-fly projection of the requested columns. Thus, ephemeral variable points to the layout that maximizes data locality. This enables transparent data transformation to better efficiency for the query at hand and lower cache pollution. Figure 3 shows an example code in C-style using ephemeral variables. The main benefit of ephemeral variable is that the software does not need to control the underlying hardware, instead, RME intercepts the CPU-oriented read requests and transparently reorganizes the data.

```
1  // layout of the full relational table
2  struct row {
3      long key;                 /* 8 bytes */
4      char text_fld1 [12];      /* 12 bytes */
5      char text_fld2 [16];      /* 16 bytes */
6      long num_fld1;            /* 8 bytes */
7      long num_fld2;            /* 8 bytes */
8      long num_fld3;            /* 8 bytes */
9      long num_fld4;            /* 8 bytes */
10 };
11 // the variable that holds the full relational table
12 struct row the_table[];
13 // the SQL query to execute
14 char* QUERY = 'SELECT SUM(num_fld1 * num_fld4) FROM the_table ↩
       WHERE key > 10';
15 // the ephemeral variable of the SQL query to execute
16 ephemeral struct column_group {
17     long key;
18     long num_fld1;
19     long num_fld4;
20  };
21 // configuring the ephemeral variable's geometry
22 struct column_group* cg;
23 cg = configure(the_table, QUERY);
24 // executing the query using the ephemeral variable
25 long sum = 0;
26 for (int i = 0; i < cg.length; i++) {
27     if (cg[i].key > 10) {
28         sum += cg[i].num_fld1 * cg[i].num_fld4; } }
```

Figure 3: An example C-style code snippet with an ephemeral variable. Line 23 configures the ephemeral variable. Upon accessing it (line 28), the hardware fetches the relevant columns as if they already exist as a column group in memory.

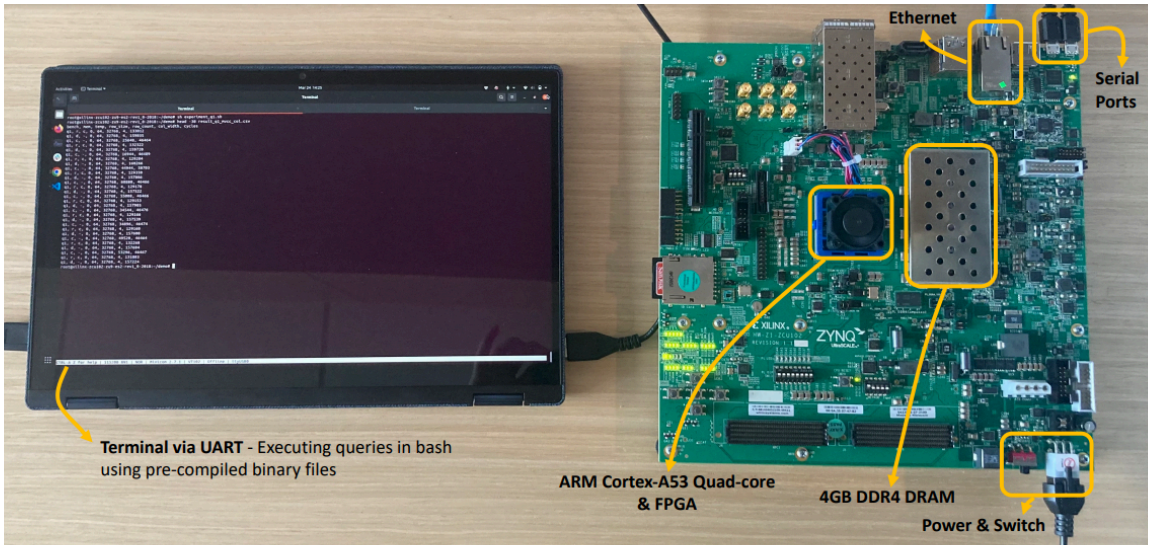

Ethernet
Serial Ports
Terminal via UART - Executing queries in bash using pre-compiled binary files
ARM Cortex-A53 Quad-core & FPGA
4GB DDR4 DRAM
Power & Switch

Figure 4: Demonstration Setup using Xilinx Zynq UltraScale+ MPSoC platform.

3 DEMONSTRATION

3.1 Target Platform

We implement RME on a Xilinx Zynq UltraScale+ MPSoC platform (ZCU102) as shown on the right side of Figure 4. This board is equipped with 4 ARM Cortex-A53 1.5 GHz cores, 4 GB DDR4 memory, and an FPGA. Each core has a private 32+32 KB L1 I+D cache and all four cores share a unified 1 MB L2 cache. The operating system is Linux 4.14, and all codes are in C/C++ and compiled using GCC 7.3.1 for AArch64. RME is integrated on the FPGA on the board and operating at 100 MHz frequency.

3.2 Demonstration Methodology

Relational Memory Benchmark. We choose a synthetic benchmark to demonstrate the usage of Relational Memory under various access patterns. Listing 1 shows the benchmark with four template queries that consist of projection, selection, aggregation, and join query on two tables where all data are in main memory.

*Q*1 is the simplest query that calculates the average of a single column. *Q*2 is a projection of *k* columns (non-contiguous or contiguous), where *k* can be varied. *Q*3 is a generalization of *Q*2 and imposes a selection of *i* columns, where *k*, *i* can be varied. Finally, *Q*4 performs a join query over two tables.

Listing 1: Relational Memory Benchmark

```
Q1: SELECT avg(A1) FROM S;
Q2: SELECT A1, A2, ..., Ak FROM S;
Q3: SELECT A1, A2, ..., Ak FROM S WHERE C1, C2, ..., Ci;
Q4: SELECT S.A1, R.A3 FROM S JOIN R ON S.A2 = R.A2;
```

Implementation. We compare the performance of Relational Memory with custom-implemented in-memory row-store (based on the Volcano-style tuple-at-a-time processing model) and column-store (following the column-at-a-time processing model).

Demonstration Setup. Throughout our demonstration, we vary the tunable parameters to highlight their impact on the performance of RME. By default, the size of each column is 4 bytes, and the size of each row is 64 bytes unless otherwise stated. The procedure to run queries using RME consists of three steps, as shown in Listing 2: populating DB, configuring RME, and then performing the query.

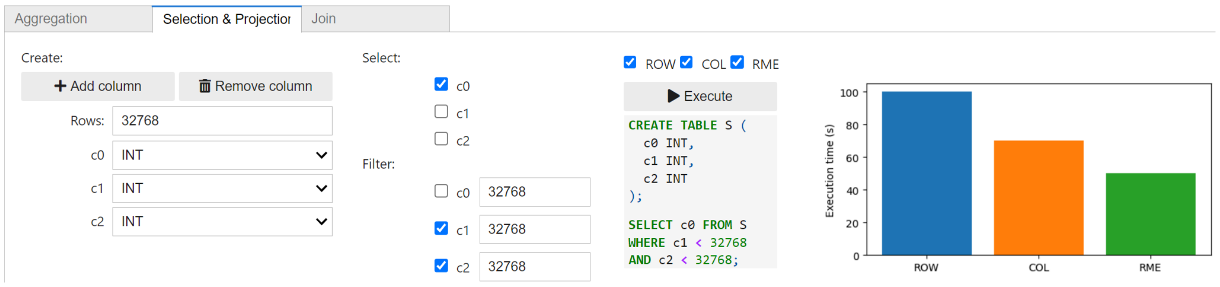
Table 1: The configurable parameters of DB generator.

symbols	options	description
S	**r** | c	row store (r) or column store (c)
r	int	row width in byte (64)
R	int	number of row counts (32)
C	int	number of columns in each row (16)
W	int	array of column width (4)
T	s | **r** | z	array of column type
m	int	minimum value for random type (0)
M	int	maximum value for random type (1000)
V	bool	supporting MVCC (false)
P	bool	print the generated table (false)

Listing 2: RME configuration

```
./db_generator -r $ROW_SIZE -R $ROW_COUNT -M $MAX
./config_rme -r $ROW_SIZE -R $ROW_COUNT -C $proj_col_num -W ↩
     $width -O $col_off -F $frame_off
./query_i -r $ROW_SIZE -R $ROW_COUNT -C $proj_col_num -W $width ↩
     -O $col_off -F $frame_off
```

<u>*DB Generator.*</u> We implement a DB generator that takes ten parameters (Table 1) from the user to generate a base table to run the queries: the layout (row store or column store), the row width, the number of rows, the number of columns in a row, the widths of each column, the types of each column (sorted, random, or zero-padded), the minimum value for the random type and the maximum value for the random type. Note that the DB generator does not support variable length since the current implementation of RME does not support variable length columns, while RME is capable of supporting arbitrary length columns. The DB generator allows controlling the query selectivity by setting the appropriate values for the minimum and maximum values. Finally, the DB generator supports MVCC by adding two timestamp fields for every row.



Create:

Rows: 32768

c0 INT

c1 INT

c2 INT

Select:

☑ c0

☐ c1

☐ c2

Filter:

☐ c0 32768

☑ c1 32768

☑ c2 32768

☑ ROW ☑ COL ☑ RME

```
CREATE TABLE S (
  c0 INT,
  c1 INT,
  c2 INT
);

SELECT c0 FROM S
WHERE c1 < 32768
AND c2 < 32768;
```

Figure 5: Demonstration interface for Relational Memory. Users will be able to select among the queries and vary different parameters to compare RME's performance with a row-store and column-store implementation.

Configuring RME. RME needs the following information for configuration: 1) the geometry of the base table, such as the address of the base raw table, the width of the row, and the number of rows in the table, and 2) the information about the query, such as the number of columns that the query needs, and the size of each desired column, and the position of each column within a row. According to this configuration setup, RME generates the descriptors to fetch the data from main memory and store the necessary parts only. The second line of Listing 2 shows the bash script to use the configuration.

Performing Queries. RME is now ready to transform the data. There can be query-specific parameters, however, the default set of parameters is identical to the configuration. The details about each query will be described in Section 3.3.

3.3 Demonstration Scenarios

We developed a web-based (Jupyter Notebook style) interface as shown in Figure 5 where conference participants will explore four demo scenarios (Listing 1) while modifying various parameters to analyze RME's performance. The interface creates a bash script (Listing 2) according to the parameters and sends it to the board for execution. The participants will be able to observe the live demonstration of the results in two ways: a graph that compares the execution time and the live signals captured using Xilinx Vivado.

S1. Overhead of Fetching Data through RME. This scenario shows the overhead of data fetching through RME by running $Q1$, which calculates the average of a single column. In addition to the row-wise and columnar accesses, we run two sets of experiments for RME: *hot* and *cold*. The *hot* case is when the desired column data is already in the data buffer inside RME, while RME needs to fetch the data for the *cold* case. We present live results.

S2. MVCC Transactions. RME supports multi-version concurrency control (MVCC) transactions through snapshot isolation. The ephemeral variable is read-only; thus RME updates the row-oriented base data by using two timestamps for each row to support multiple versions. The first timestamp indicates the beginning of the validity of the row, while the second timestamp is set to mark the end of its validity. Here, we use $Q2$ to demonstrate the MVCC transactions compared to row-wise and columnar accesses.

S3. Scalability of RME. RME supports data transformation of arbitrary data size even though the size of data SPM is only 2 MB. $Q3$ is executed to evaluate the scalability of Relational Memory while increasing the data size up to 1 GB. Note that the row-wise and columnar accesses perform the query at once regardless of the data size. However, RME needs to perform an *invalidation* process whenever the data SPM is full which is done within a single clock cycle. In this scenario, the participants will observe how RME can handle arbitrary data size without affecting its performance.

S4. Join Queries. This scenario performs $Q4$ to highlight that RME supports join queries. We implement join using a state-of-the-art hash-based join algorithm with a single-pass hash table generation. Note that half of the entries of the outer relation have a match in the inner relation. In addition, the CPU cost for $Q4$ is notably high compared to $Q1 - Q3$ due to hash calculation. We also demonstrate the benefit of using RME for CPU and data movement separately.

4 CONCLUSION

In this demonstration, we show how to interact with Relational Memory Engine (RME), an on-the-fly vertical partitioner that allows to access optimal data layout while keeping the base data in row format only. Participants will be able to interact with RME implemented in a real PS-PL platform, configure it, write multiple queries with it and analyze the performance comparison with respect to row-store and column-store via a web-based interface.

ACKNOWLEDGMENTS

This work is funded by a RedHat Research Incubation Award, a RedHat Research Award, a Cisco gift, and partially supported by the National Science Foundation (NSF) under grant number IIS-2144547, CCF-2008799 and CNS-2238476.

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.

[2] 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.

[3] 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.

[4] 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.

[5] Tarikul Islam Papon, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Fabric: Transparent Data Transformation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

[6] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 66–79.

[7] Neil C Thompson and Svenja Spanuth. 2021. The decline of computers as a general purpose technology. *Commun. ACM* 64, 3 (2021), 64–72.