Relational Fabric: Transparent Data Transformation [Vision]

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

***Abstract*—A key design decision for data systems is whether they follow the row-store or the column-store paradigm. The former supports transactional workloads, while the latter is better for analytical queries. This decision has a profound impact on the entire data system architecture. The multiple-decade-long journey of these two designs has led to a new family of hybrid transactional/analytical processing (HTAP) architectures. Several efforts have been proposed to reap the benefits of both worlds by proposing systems that maintain multiple copies of data (in different physical layouts) and convert them into the desired layout as required. Due to data duplication, the additional necessary bookkeeping, and the cost of converting data between different layouts, these systems compromise between efficient analytics and data freshness. We depart from existing designs by proposing a radically new approach. We ask the question:**

"What if we could access any layout and ship only the relevant data through the memory hierarchy by transparently converting rows to (arbitrary groups of) columns?"

To achieve this functionality, we capitalize on the reinvigorated trend of *hardware specialization* (that has been accelerated due to the tapering of Moore's law) to propose *Relational Fabric*, a near-data vertical partitioner that allows memory or storage component to perform on-the-fly transparent data transformation. By exposing an intuitive API, Relational Fabric pushes vertical partitioning to the hardware, which has a profound impact on the process of designing and building data systems. (A) There is no need for data duplication and layout conversion, making HTAP systems viable using a single layout. (B) It simplifies the memory and storage manager that needs to maintain and update a single data layout. (C) It reduces unnecessary data movement through the memory hierarchy allowing for better hardware utilization, and ultimately better performance. In this paper, we present Relational Fabric for both memory and storage. We present our initial results on Relational Fabric for in-memory systems and discuss the challenges of building this hardware, as well as the opportunities it brings for simplicity and innovation in the data system software stack, including physical design, query optimization, query evaluation, and concurrency control.

I. Introduction

OLTP vs. OLAP vs. HTAP. The popularity of large-scale real-time data analytics has soared over the past few years [54], [57], further fueled by the advent of new technological trends like 5G, Internet-of-Things, and cloud computing [18], [29]. Database management systems (DBMS) now need to perform both Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) since many applications need to analyze fresh data. However, traditional OLTP and OLAP systems are very different – OLTP systems generally use a row-oriented data layout to optimize for write-intensive workloads and point queries. In contrast, OLAP systems use a columnar layout to optimize for read-only analytical queries. Hybrid Transactional/Analytical Processing (HTAP) [57] systems aim to maintain a single data store that offers data freshness and efficient analytics on that same data set. HTAP systems attempt to bridge the OLAP and OLTP requirements by maintaining multiple copies of data in different formats [14], [39], [61] or converting data between different layouts [9], [12], [45], [50], [70]. However, recent research on such hybrid layouts [6], [12], [21], [40] shows the optimal layout of a query is often neither a columnar nor a row-oriented one. Such approaches carry extra complexity, either to convert data between layouts or to manage multiple versions and copies of the data, while not always being able to offer the optimal layout.

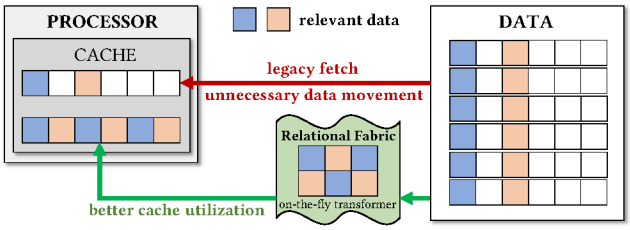


Fig. 1: Relational Fabric removes unnecessary data movement by transparent on-the-fly data transformation.

Hardware Specialization On The Rescue. Due to the historically exponential growth of processor speed, the idea of *hardware specialization*, although recurring every few years, has not been able to fulfill its true potential. As Moore's law slows down and the data processing needs keep growing exponentially, hardware specialization is becoming a feasible, and more scalable alternative to general-purpose computing [34], [75]. In this *post-Moore* era, this has been further accelerated by the advancements in reconfigurable logic and the rise of large-scale data processing applications. Recent technological developments (e.g., Google's TPU [30], Microsoft's Catapult [53]) show that specialization is here to stay [34].

Idea: Transparent Data Transformation

What if we can access any arbitrary data layout using near-data processing via specialized hardware?

In other words, *"what if the optimal data layout is always physically available?"*. This will eliminate the need to keep multiple layouts and we can perform efficient analytics over the fresh data without converting between different layouts. If the underlying specialized hardware accesses only the relevant data (without accessing unnecessary data and without paying a tuple reconstruction cost) while maintaining a single layout, it will blend the benefits of row-stores and column-stores. It will offer *effortless locality*, alleviating the need for separate row-store and column-store query engines, decoupling the *physical* data layout from the *processing* data layout.

Our Vision: Relational Fabric

A lightweight specialized hardware fabric that allows accessing arbitrary data layouts from memory or storage.

We propose *Relational Fabric*, a new lightweight specialized hardware fabric that accommodates queries to access arbitrary column groups from memory-resident base data without requiring any data duplication. The base data is stored in a row-oriented physical layout (Figure 1), to allow efficient data ingestion and updates, read-only queries can quickly access only the relevant column groups (or the entire row, if needed) using the underlying machinery. To do this, Relational Fabric exposes a carefully designed API, termed *ephemeral columns* that enables accessing arbitrary *data geometries* (i.e., any subset of data from relational tables) using simple abstractions. This API 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. This, in turn, supports both efficient column- and row-oriented accesses while minimizing CPU cache pollution with unnecessary attributes. Relational Fabric has three major benefits:

- ✓ **Low Data Complexity:** It allows efficient HTAP processing while maintaining only one layout of the data. No need to propagate changes to multiple data copies or to convert data among different layouts.
- ✓ **Low Software Complexity:** It reduces data system software complexity. No need to maintain different execution engines. Rather, the execution engine can always assume that only the relevant data will be accessed.
- ✓ **Efficient Hardware Utilization:** It provides *effortless locality* via shipping only relevant data through the memory hierarchy, alleviating unnecessary data movement, and providing better cache and processor utilization.

As a first instance of Relational Fabric, we have developed *Relational Memory* [63] that utilizes recent advancements in programmable logic [64], and pushes projection to the hardware (§II). The API of Relational Memory is a simple, lightweight programming abstraction, termed *ephemeral variables*, enabling the CPU to access arbitrary data geometries. Relational Memory exploits the inherent parallelism of memory cells to efficiently access data in scattered locations, and uses programmable logic to reorganize and compact it on the fly before pushing it to the CPUs, thus improving locality.

Simplifying the Data Systems Software Stack. With Relational Fabric in place, the data system software stack can be significantly simplified. A data system that makes good use of the Relational Fabric would have to drastically simplify its physical design and query optimization components, while the query evaluation engine would now be able to make the most of code generation. Other components would also be significantly affected, especially the transactions manager that would have to implement a multi-version concurrency control (MVCC) approach, and the compression algorithms that should be compatible with scattered data accesses.

In this paper, we present our vision of building *Relational Fabric*, along with the challenges and opportunities of innovation in data systems and some open research questions to fully realize this vision. We begin by discussing how to access arbitrary data geometries and present our initial work on Relational Memory and ephemeral variables (§II). The Relational Fabric vision has several *opportunities for simplicity and innovation* across the data systems stack (§III):

- **Simplify Physical Design:** Relational Fabric will grossly simplify the physical design process. There is no need for creating (physical) vertical partitions anymore. Further, indexes will mostly be useful for workloads with point queries and updates, since range queries can be very efficiently evaluated with column-group accesses. Overall, through the Relational Fabric any layout can be achieved via on-the-fly data reorganization.
- **Simplify Query Optimization:** One of the key challenges with query optimization is that it is a combinatorial process that has to search a vast space. Relational Fabric increases the search space by allowing access to any data layout, however, this essentially removes any search constraints in that space. Hence, instead of *solving a combinatorial problem*, we can now *construct* the fastest solution.
- **Efficient MVCC:** The natural way to implement concurrency control using Relational Fabric is MVCC, where there is one source of truth (the base data in row-oriented format), and the ephemeral columns access the correct data using timestamp information associated with MVCC. A big win for Relational Fabric is that it implements timestamp comparisons in hardware, leading to a simple and good-performance implementation of MVCC.

Building the Relational Fabric hardware requires bringing together software and hardware expertise and benefit from the tight integration of programming systems and programmable logic which is increasingly gaining momentum (§IV). Our preliminary Relational Memory design is implemented in such a tightly integrated platform. We further outline our vision for integrating data transformation with the memory controller and extending the processor's ISA. The ultimate goal is to be able to use Relational Fabric without needing deep expertise in hardware/software co-design.

Outline. The remainder of the paper is organized as follows. Section II presents the concept of accessing arbitrary data geometries in detail. Section III discusses the implications on

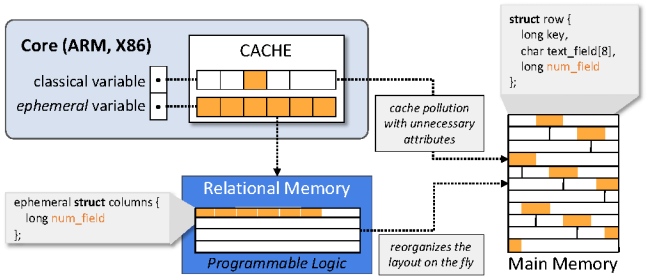


Fig. 2: Architecture and data flow of Relational Memory, an in-memory instance of Relational Fabric that pushes projection closer to the in-memory data.

the data management software stack, Section IV discusses how to build the Relational Fabric hardware, and Section V presents our early results. Finally, Section VI discusses related work, Section VII outlines aspects of the vision that warrant further investigation, and Section VIII concludes.

II. Accessing Arbitrary Data Geometries

Motivation. When considering a row-oriented relational table, accessing a specific (group of) column(s) leads to a strided access pattern, where a few bytes (size of accessed columns) are accessed every hundred or thousand bytes of data (size of the row). This is inefficient since each new row always pulls an entire cache line from memory resulting in lower cache utilization and more data movement than what is required (Figure 1). Further, strided accesses with large strides are not handled well by hardware prefetchers [10]. Column-stores avoid strided accesses by storing each attribute separately. This results in accesses with high locality since only data items strictly required for the final computation are transported from the main memory. However, it also comes at the cost of increased tuple reconstruction cost for queries with high projectivity [1], and it is an inefficient layout for inserts and deletes. Instead, we want to achieve *effortless locality* for queries with any projectivity without having unnecessary data movements or any tuple reconstruction cost.

Relational Fabric. To achieve this, we propose to utilize specialized hardware to perform transparent on-the-fly data transformation while the base data is stored in a row-oriented layout. This allows efficient updates and inserts on the base data while analytical queries may access the desired columns through Relational Fabric. Data transformation and selection operators like *projection* and *selection* can be pushed closer to data to reduce unnecessary data movement.

For memory-based systems, the hardware will sit between memory and the processor, intercept the CPU-originated requests and transparently reorganize the data on-the-fly, making it like the desired data layout already exists in memory. For disk-based systems, the hardware will sit between memory and storage and will be able to transparently reorganize the data on-the-fly, making it like the desired data layout already exists in storage. Operating closer to data enables us to utilize the underlying device parallelism (either within different memory

```
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
12 // the variable that holds the full relational table
13 struct row the_table[];
14
15 // the SQL query to execute
16 char* QUERY = 'SELECT SUM(num_fld1 * num_fld4) FROM ↩
       the_table WHERE key > 10';
17 // the ephemeral variable of the SQL query to execute
18 ephermeral struct column_group {
19     long key;
20     long num_fld1;
21     long num_fld4;
22  };
23 // configuring the ephemeral variable's geometry
24 struct column_group* cg;
25 cg = configure(the_table, QUERY);
26
27 // executing the query using the ephemeral variable
28 long sum = 0;
29 for (int i = 0; i < cg.length; i++) {
30     if (cg[i].key > 10) {
31         sum += cg[i].num_fld1 * cg[i].num_fld4; } }
```

Fig. 3: The ephemeral variable is configured in line 25. Once the CPU accesses it (line 31), the machinery starts fetching the desired columns to the CPU as if they already exist as a column group in memory.

banks [33], [47] or within storage devices like flash-based SSDs [16], [58].

To ensure ease of programmability, the specialized hardware is used via a simple abstraction that allows the data system engineer to request the desired column groups (in case of projection) or rows (in case of selection). For the memory-based design, it will require compiler support to make this fully transparent to the system developer. In addition, in order to make the memory fabric easy to integrate we envision integrating it into the memory controller and modifying the instruction set architecture (ISA). For the disk-based design, the system developer will use a dedicated API that allows to directly request from the proposed machinery to fetch the desired columns from storage. This API will directly feed into the scan operator sending only the relevant data for further query processing. We now discuss our first instance of Relational Fabric and its API that targets in-memory data systems and is termed *Relational Memory*.

Relational Memory (RM). To offer contiguous access to a specific column (or column-group) in memory, we built Relational Memory (RM) [63], which leverages the PLIM paradigm [64] to *create references to data that does not exist in main memory*, which *the CPU can use as if the data does exist* in main memory. In other words, RM enables accessing the same content in main memory under different strides, but it can be accessed as if it was stored contiguously from the CPU's perspective. 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. Furthermore, operating closer to the data allows to exploit the inherent parallelism of memory cells. Figure 2 shows a high-level diagram of the proposed design. Relational Memory is located in programmable logic sitting between the memory and the processor. Upon receiving a request, the engine transforms on-the-fly rows to any desired combination of columns. The processor directly accesses data in the optimal layout through an abstraction called *ephemeral variables*, which we discuss next.

Ephemeral Variables. RM aims to provide the optimal layout for any possible queries; thus, RM executes projection and tuple reconstruction in hardware, and the reconstructed tuples are accessible via *ephemeral variables*, a special type of variables that identifies a specific subset of columns to access. *Ephemeral variables* create memory *aliases* to expose non-contiguous content as if they were contiguous. These transient 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. Figure 3 shows an example of using ephemeral variables.

This abstraction of *non-materialized in-memory aliases of column-group accesses* grossly simplifies the data management software stack, as we will discuss in Section III. While in our current prototype ephemeral variables are supported by low-level macros, our long-term vision is to add full compiler support by integrating Relational Memory into the memory controller and expanding the ISA, which we discuss in more detail in Section IV. This will allow RM to benefit any data systems with minimal engineering effort.

III. Implications on Data Systems Architecture

Relational Fabric can significantly simplify the data system software stack. We now discuss the opportunities for simplicity and innovation across the data systems software stack. Figure 4 shows the database components that will be affected by Relational Fabric (green background).

A. Physical Design

The physical design process of a database entails (i) the mapping from real-world data to relational data (tables, rows, and columns), and (ii) decisions on how to physically store the relational table regarding normalization, data layout, data partitioning, and indexing. While the first part depends on the nature of the application, the second part aims to optimize performance while respecting some constraints (e.g., space amplification) using workload knowledge. Relational Fabric can radically simplify the second step of physical design because it can allow access to different data geometries without needing to physically store data in that format. We now discuss how the key decisions can be simplified.

Data Layout and Vertical Partitioning. One of the most impactful physical design decisions is the data layout: a row-oriented, a column-oriented, or a hybrid layout. This decision is driven by the application requirements since a row-oriented

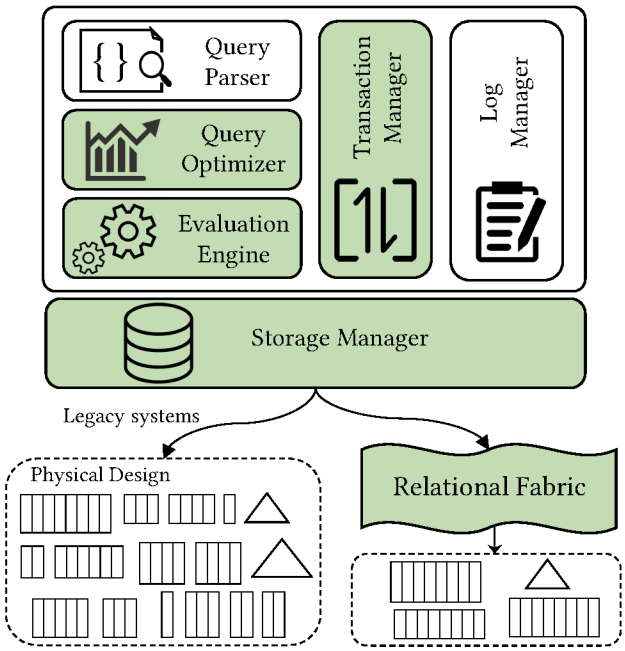


Fig. 4: Relational Fabric enables opportunities for innovation across the entire data system architecture. It simplifies physical design, provides more options for query processing and supports efficient multi-version concurrency control.

layout is a good fit for transactional applications, while a columnar layout performs well for analytical systems. To bridge the analytical and transactional requirements, hybrid systems often maintain different engines for ingestion and analytics, paying the cost of data layout conversion, duplication, and the necessary additional bookkeeping. Both traditional and hybrid systems use the expected workload to form educated decisions about storing vertically partitioned data. The goal is to collocate columns that are frequently accessed together to reduce unnecessary data movement.

Relational Fabric simplifies the physical design process. Since the query evaluation engine can always request any arbitrary column groups, there is no need to restrict the physical design space to a predefined set of vertical partitions. The base data is stored in a row-oriented format and can efficiently facilitate updates and ingestions, and any analytical query can be executed using the optimal column group, i.e., the column groups that are required by the query. Thus, Relational Fabric eliminates the need for creating physical vertical partitions since the base data can be on-the-fly vertically partitioned with the help of specialized hardware. Note that indexes can still be defined on the base (row-oriented) data. However, the usefulness of indexes is now smaller, since range queries can be efficiently evaluated with columnar accesses, so indexes should be used for point queries and point updates. While Relational Fabric does not necessitate any major changes in the storage manager, it simplifies its operation, as it only needs to maintain a single copy of each relation's data.

Horizontal Partitioning. Contrary to vertical partitioning that can happen on-the-fly using Relational Fabric, horizontal par-

titioning decisions would still need to be evaluated at physical design time. While Relational Fabric does not affect horizontal partitioning, it can be efficiently combined with it, along with sharding which horizontally partitions data based on a sharding key. Another functionality that Relational Fabric can integrate is to handle the communication with storage devices while exposing its simple *ephemeral columns* API to the query. That way, the data system can request the desired column group on a sharding key range, and the Relational Fabric will directly return the corresponding data to the query.

Indexing. Indexes help accelerate access times, ensure uniqueness, and allow sorting and clustering. In general, row-store systems employ indexes that are useful for selection queries and data updates. Column-store systems (and some recent row-store systems) [44], [46] use column projections as a special type of index. The strength of Relational Fabric is that it makes such projections possible without having to materialize them. This has deep implications on the data systems architecture because the query engine may access the data at query time using either the base row-oriented data, an index (if it exists), or the desired columns projected from Relational Fabric. While indexes will be mostly beneficial for workloads with point queries and updates, Relational Fabric allows for efficient range queries due to the arbitrary column-group accesses.

B. Query Optimization and Query Evaluation

When a query is submitted to the system, it is first processed by the query parser via lexical, syntactic, and semantic analysis. Then the initial query plan is processed by the query optimizer to find the best query plan to execute, which is, in turn, submitted for execution to the query evaluation engine. Relational Fabric will not affect the query parser, however, the optimizer and the evaluation engine components will have a significant impact. Row-oriented query execution models execute query plans one-row-at-a-time, which offers good performance for OLTP queries. In contrast, most column-store systems use vectorized execution that progresses through the query plan via processing batches of column data [41], [43]. While Relational Fabric maintains the base data in row format, it makes columns available to the processor on the fly, and, thus, it can naturally support efficient vectorized query execution. A key challenge with query optimization is the combinatorial process of searching a vast space – the DMBS has to select the most efficient evaluation plan based on the cost of each plan. While Relational Fabric seems to increase the search space by enabling access to arbitrary data layouts, it also allows every query to always use the best layout, and overall simplify the problem, by replacing the *combinatorial search* with the *construction* of the query plan that accesses the optimal data fragments. This opens a whole new avenue of research with the potential to speed up query processing: (i) generate the fastest query plan, (ii) revise existing cost models considering Relational Fabric, and (iii) re-evaluate classical single-column, multi-column partitioning cost models.

Code Generation. Adaptive systems like Hyper, H_2O, Actian Vector, Hekaton, MemSQL and others [6], [20], [40], [51], [70] examine the query and decide how data will be accessed by evaluating alternative access plans. The appropriate code is generated by considering the buffered available data layouts. One disadvantage of this approach is the requirement to compile code on the fly which is alleviated by buffering compiled code fragments. The development of Relational Fabric aids code generation in two ways. First, Relational Fabric does not require to buffer different layouts since any arbitrary layout can be accessed on the fly. Second, since data layouts are not buffered, Relational Fabric can buffer more code fragments and reuse previously compiled code fragments more aggressively. The query optimizer can now also consider various factors like which code fragments are buffered and which indexes are available. One key opportunity of innovation through Relational Fabric 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 while working on the same base data.

C. Concurrency Control

The transaction manager is responsible for concurrency control. Relational Fabric can naturally support multi-version concurrency control (MVCC). While the base data is in row format, Relational Fabric offers native access to arbitrary data geometries through ephemeral columns. For example, in our in-memory implementation of Relational Memory, we use ephemeral variables to access column groups. We consider all ephemeral variables (or the respective API) as *read-only* columns or column-groups that accelerate analytical queries. The row-wise base data is marked as *read/write* and updates are handled by appending new rows to this base data. For updates and deletion, Relational Fabric uses two timestamp fields for every row to support multiple versions. The first timestamp is set when a row is inserted to mark the beginning of its validity, while the second timestamp is set upon row deletion or replacement by a newer version, marking the end of its validity. Every time the API is accessed, it generates the column groups that contain the valid rows at the time of the query. A key advantage of this approach is that the timestamp comparison can be implemented in hardware, making this implementation simple and performant. By offering the optimal layout and using the timestamps to ship only valid data, Relational Fabric supports MVCC transactions through snapshot isolation.

D. Compression

The proposed design stores the base data in a row-oriented format hence it can benefit only from specific types of compression. General compression algorithms of the LZ family [85] are frequently used by row-oriented systems, however, they are not a natural fit for Relational Fabric since they require fully decompressing your data before you can access separate columns. Delta, dictionary, and huffman encoding for compression which are popular among state-of-the-art column stores [2], [3], [86] are easily supported by Relational Fabric. Note that these schemes can be used in row-oriented data,

and hence, they can benefit any groups of columns requested by ephemeral columns. However, the compression schemes under the run-length encoding family cannot be used out of the box. In contrast to dictionary and delta encoding, RLE has an expensive decoding step and relies on data, but it is still quite popular among column stores. More research is required to find compression techniques that can benefit both row-oriented and columnar data and allow for direct operation on compressed data.

IV. Building Relational Fabric

The key feature of Relational Fabric is that a specialized hardware component transforms data on-the-fly. This section presents the implementation details of our first in-memory Relational Fabric instance, termed Relational Memory (RM). We also discuss extending hardware support to more operators, pushing the logic further into the memory controller, and Relational Fabric for storage devices.

A. Implementing Relational Memory

RM is an FPGA-based data transformation engine that sits between the processor and the memory and converts data layouts on-the-fly as shown in Figure 2. Data transformation in RM 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. The developed hardware performs following four key operations: (1) RM receives 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, (2) RM communicates with memory via an AXI bus [11] and assembles multiple entries into a single *packed* cache line to be sent to the processor, in the meantime, (3) RM captures the CPU requests and (4) transfers the reorganized data upon availability. This abstraction creates *non-materialized aliases of column-groups* which pushes arbitrary subsets of columns to the memory hierarchy. Hence, RM supports both efficient column- and row-oriented accesses while minimizing CPU cache pollution with unnecessary attributes.

B. Pushing Other Relational Operators

Relational Memory is the first instance of a new class of data systems architectures. Implementing projection in hardware lays the groundwork for pushing other relational operators to the hardware as well. The Relational Fabric philosophy is that new hardware designs will be adopted if they are simple and general. In other words, a very application-specific design is hard to make its way to mass production. With that in mind, we propose to further reduce unnecessary data movement by transparent on-the-fly data transformation using minimal hardware complexity, by pushing selection and aggregation in the hardware. Both operations are general enough in the sense that they are part of other applications (like operating on matrices and tensors) and have the potential to offer even larger data movement reduction benefits. Implementing selection in Relational Fabric will further alleviate the need for indexes altogether, while aggregation will help both relational and matrix operations. In this design, the ephemeral variables will contain only the required data or the aggregation result, which will be passed through the memory hierarchy ensuring minimal data movement while maintaining the hardware complexity low.

C. Pushing RM Further: Relational Memory Controller

Following the aforementioned philosophy, we are developing another instance of Relational Fabric, termed Relational Memory Controller (RMC) where we place RM further closer to memory. Integrating RM into a memory controller has the potential to become a game-changer as it will allow for easy adoption of the RM design with a minimal development effort. Further, pushing RM into the memory controller maximizes its benefits, since it has low-level access to the actual memory DIMMs. State-of-the-art memory controllers handle complex interfacing with DDR memories while guaranteeing reliable performance. However, memory controllers cannot fully exploit the capabilities of DDR memory chips since they have no information about the target workloads or environments. By integrating RM with a memory controller, just enough semantic information about the access patterns will make it to the hardware, making it possible to fully exploit the capabilities of DDR memory chips, thus offering superior performance and further reducing unnecessary data movement.

Extending the ISA as an RMC Interface. An Instruction Set Architecture (ISA) is the abstraction between hardware and software. The ISA guarantees that the resulting binary code correctly executes regardless of the toolchain; thus, it helps developers to write and debug software more efficiently [67]. Therefore, integrating RM with ISA, such as RISC-V [13], [78], provides a more valuable interface. The benefits of using RM via ISA are two folds: (1) it will provide a simple interface with no need to understand the details of the underlying hardware and (2) it will simplify the code generation process during the compile time. Thus, RMC along with an ISA extension provides a simple API that can further be beneficial for SQL queries (or other applications benefiting from data transformation) [66], [80].

D. Implementing Relational Storage

Following our discussion about building Relational Memory, we propose to develop Relational Fabric in storage devices. Near-storage computation is more challenging than near-memory computation because traditionally storage devices are incapable of performing logic. However, recent modern storage devices like SmartSSD [59] and OpenSSD [15] have processing power that can be exploited to achieve this. We call this approach of pushing computation to storage Relational Storage (RS). RS can be directly implemented in a specialized storage device (i.e., in OpenSSD or SmartSSD) or a programmable logic (i.e., FPGAs), similar to our Relational Memory approach. In contrast to RM, it is possible to push other operators like selection and aggregation by utilizing the processing capabilities of in-storage custom logic. Even

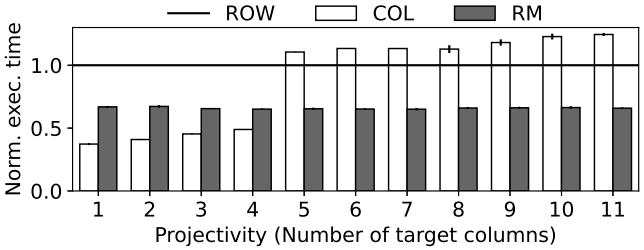
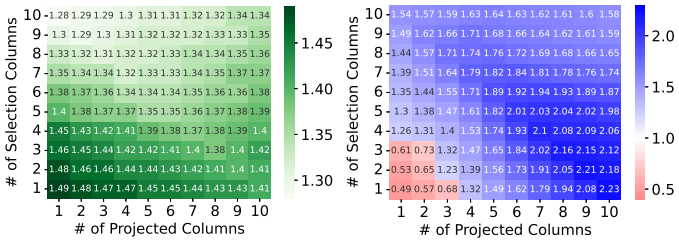
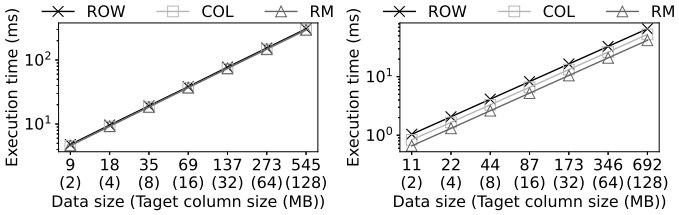


Fig. 5: RM outperforms row-wise memory accesses irrespectively of projectivity, while RM shows better performance than columnar accesses for projecting more than 4 columns.



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

Fig. 6: (a) RM always outperforms in-memory row access. (b) RM dominates when the total number of columns grows larger (> 4), while columnar accesses achieve better performance when the total number of columns is small (≤ 4).



(a) $Q1$ (b) $Q6$

Fig. 7: RM shows better performance than direct row-wise or columnar accesses in practical queries such as TPC-H $Q1$ and $Q6$ regardless of the data size.

decompression can be done on-the-fly along with data transformation [17]. In a similar manner, exploiting the internal parallelism of the storage device [58] can enhance performance. Furthermore, the software stack will be redesigned to take advantage of near-storage computation for better query processing and optimization in contemporary storage devices.

V. Early Results on Relational Memory

We now present selected experimental results of RM showing that it outperforms direct row-wise and direct columnar accesses by offering the optimal layout to any query [63].

Target Platform. The full-stack prototype of RM is implemented on a Xilinx Zynq UltraScale+ MPSoC platform [82] which consists of heterogeneous Systems-on-Chip (SoC) where a traditional processing system (PS) is tightly associated with a programmable logic (PL), i.e., an FPGA. The PS equips 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. PL side, RM prototype, is constrained to 100 MHz. In order to compare the performance of RM to the row-store (ROW) and the column-store (COL), 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.

RM Shines for Queries with High Projectivity. In the first experiment, we vary the projectivity from 1 to 11 columns for 4-byte wide columns and 64-byte wide rows, as shown in Figure 5. For any projectivity, RM outperforms direct row-wise accesses since RM provides the optimal layout that minimizes cache pollution. When the projectivity is low (≤ 4), columnar accesses are faster since the tuple materialization cost is still small and the prefetcher can efficiently support up to four parallel sequential accesses. As projectivity becomes larger than four columns, however, RM starts to outperform direct columnar accesses due to the tuple reconstruction cost.

RM Offers Optimal Projection-Selection Queries. The experiment shown in Figures 6a and 6b compares the performance of RM with direct row-wise and columnar accesses while varying the number of columns in a projection and selection query. The number of projected columns (x-axis) and the number of columns used for selection (y-axis) range from 1 to 10 columns. Figure 6a shows the speedup of RM compared to the direct row-wise accesses. Similarly to the previous experiment, RM consistently outperforms the direct row-wise access by 1.3-1.5×. In contrast, direct columnar access achieves better performance than RM when the number of columns used for projection and selection is less than four as shown in the lower left corner of Figure 6b. As the number of columns in a query increases, RM outperforms the columnar accesses. Overall, RM achieves better performance than direct row-wise accesses for any number of target columns, while RM outperforms a columnar layout only when the number of target columns is large enough (> 4).

RM Shows Stable Performance for Practical Queries. In order to evaluate RM in a practical environment, we execute $Q1$ and $Q6$ from TPC-H [76] while varying the data size. RM supports arbitrary data sizes even with a small data memory of 2 MB on the FPGA by refilling it whenever it is full. Figure 7 shows the running time of $Q1$ and $Q6$ on tables from 11 MB to 692 MB. Since we choose the data size based on the size of target columns (shown in the parentheses of x-axis), the range of data sizes varies for $Q1$ and $Q6$. For $Q1$, the execution time is similar for all layouts, as shown in Figure 7a. This is because executing CPU-intensive operations in $Q1$ dominates the data movement cost. On the other hand, for queries such as $Q6$ where data movement is the bottleneck, RM accelerates the execution time by offering the optimal layout (Figure 7b).

Overall, our proof-of-concept prototype Relational Memory shows better or comparable performance compared to in-memory row- or column-store for various queries by offering data transformation on-the-fly (more experimental results are available in our conference paper [63]). These experimental results of RM further fuel our vision for Relational Fabric.

VI. Related Work

Hybrid Layouts. Many HTAP systems like SAP HANA [27], Oracle TimesTen [45], MemSQL [70], BatchDB [49], and L-store [65] follow the *one size does not fit all* rule [73], hence, they use the row-format to ingest data and then convert it to columnar-format for analytical processing [57]. The *optimal* layout is more often neither a column-store or a row-store [6]. On the other hand, systems like H_2O [6], Hyper [40], Peloton [12], and OctopusDB [21] use adaptive layouts depending on the query patterns. All these systems need to store multiple layouts of the data and convert between formats which increases the complexity, materialization overhead, and maintenance cost.

Hardware Specialization. There have been many efforts to utilize specialized hardware for data management systems [26], [36]. We categorize the developed specialized hardware by its objectives. The first line of specialized hardware is to accelerate particular DBMS operators such as selection [74], aggregation [19], compression [60], decompression [25], data partitioning [38], sort [84], group by [4], and join [32], [83]. Secondly, we classify attempts to offload the SQL query itself or the subset of queries [55], [56], [71], [79], [80], [81]. Due to the inflexible nature of hardware, these approaches' main limitation lies in supporting ad-hoc queries. A third class is query accelerators accessing non-local memory aiming to reduce data movement [5], [8], [28], [42], [62], [72].

Contrary to the aforementioned related work or the Processing-In-Memory (PIM) approach [48], [69], the Relational Fabric paradigm does not aim to implement complex logic near memory/storage, nor to change the physical memory/storage hardware (e.g., memory or flash cells). Rather, Relational Fabric sits between the query execution engine and the data, and offers a light-weight layer that performs on-the-fly transparent data transformation into the optimal layout for the query in question without materializing it. Our first Relational Fabric instance, RM, sits between the CPU and memory and transparently transforms data into the optimal layout that does not exist in main memory. Therefore, any ad hoc queries can be accelerated with no data duplication. Furthermore, RM does not require any modification of the memory hierarchy unlike PIM and is fully implemented on commercially available platforms [7], [24], [35], [52], [82].

To develop Relational Fabric for storage devices, we capitalize on recent advancements in computational SSDs (OpenSSD [15], SmartSSD [59]). These SSDs have processing power in the flash controller that allows programmability which can be utilized to enable highly efficient SSD execution [23]. There have been several works on performing near-data processing in SSDs [22], [31], [37], [68], [77] leveraging their computational capability which can also aid the development of Relational Fabric in modern storage devices.

VII. Open Questions

In addition to the opportunities for simplicity and innovation discussed in this paper, the Relational Fabric vision has several *open research challenges* that require further investigation.

Q1. *Is data transformation (projection) enough?* Relational Fabric is a layer that offers transparent and efficient projection that leads to the benefits we discussed above. Further, data transformation has great potential for other data-intensive applications over multi-dimensional data (matrix/tensor slicing and vectorized operations on matrix/tensor slices). In addition, there have been several recent efforts to implement more complex logic near or within memory. We purposefully avoid this path because it increases the hardware complexity and specialization, making it less general and, thus, to our understanding, less appealing for real-life use and deployment. However, it remains an open question whether more logic can be implemented between the memory and the processor. Overall, our thesis is that any added logic should benefit many different applications to be ultimately viable.

Q2. *How does Relational Fabric interact with compression?* While delta and dictionary compression schemes can be used as a starting point, we also believe it is worth investigating new compression schemes that can be applied to row-oriented data and allow for on-the-fly vertical partitioning and potentially allow for operating on compressed data.

Q3. *Can you have Relational Fabric both on storage and in memory?* The vision we outline assumes that the Relational Fabric is implemented either in memory on storage, depending on the use-case. However, a scheme that uses Relational Fabric in both storage and memory may also be interesting. Consider that the two fabrics may play different roles. For example, the storage one can convert from compressed columns to rows in memory, and the in-memory one can allow the processor to access arbitrary column groups. We believe that more investigation in this direction is warranted.

VIII. Conclusion

In this paper, we present our vision of Relational Fabric, a new lightweight specialized hardware fabric that offers *effortless locality* by accessing arbitrary data layouts from row-oriented base data without any data duplication. Relational Fabric will simplify data and software complexity, and it will enable efficient hardware utilization and true HTAP processing. We outline the principles, goals, and impact of Relational Fabric, and as a proof-of-concept, we present its first instance, Relational Memory that uses reprogrammable hardware to implement logic between the memory and the processor. Relational Memory on-the-fly convert rows to arbitrary groups of columns, alleviating the need to vertically partition data. We further outline the necessary steps toward building Relational Fabric in memory, discuss its opportunities for innovation in data systems architecture in physical design, query processing, and concurrency control, and some open questions that require further research. In addition, we discuss building Relational Fabric in computational SSDs by developing Relational Storage. Developing Relational Fabric in memory and storage has the potential to be a paradigm shift where different specialized hardware components (in memory and storage) can synergistically turn data processing more efficient, scalable, and resource-efficient for data-intensive applications.

References

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented Database Systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, 2009.

[2] D. J. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The Design and Implementation of Modern Column-Oriented Database Systems," *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[3] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating Compression and Execution in Column-oriented Database Systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006, pp. 671–682.

[4] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras, "FPGA-accelerated group-by aggregation using synchronizing caches," in *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2016, pp. 11:1—-11:9.

[5] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal, "Designing Far Memory Data Structures: Think Outside the Box," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019, pp. 120–126.

[6] I. Alagiannis, S. Idreos, and A. Ailamaki, "H2O: A Hands-free Adaptive Store," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014, pp. 1103–1114.

[7] G. Alonso, T. Roscoe, D. Cock, M. Ewaida, K. Kara, D. Korolija, D. Sidler, and Z. Wang, "Tackling Hardware/Software co-design from a database perspective," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2020.

[8] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the EuroSys Conference (EuroSys)*, 2020, pp. 14:1—-14:16.

[9] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki, "The Case For Heterogeneous HTAP," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[10] ARM, "Arm Cortex-A53 MPCore Processor Technical Reference Manual," Tech. Rep., 2018. [Online]. Available: https://developer.arm.com/documentation/ddi0500/j

[11] ——, "AMBA AXI and ACE Protocol Specification," Tech. Rep., 2019. [Online]. Available: https://developer.arm.com/documentation/ihi0022/h

[12] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016, pp. 583–598.

[13] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[14] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer, "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)*, 2015.

[15] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The Linux open-channel SSD subsystem," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019, pp. 359–373.

[16] F. Chen, B. Hou, and R. Lee, "Internal Parallelism of Flash Memory-Based Solid-State Drives," *ACM Transactions on Storage (TOS)*, vol. 12, no. 3, pp. 13:1–13:39, 2016.

[17] X. Chen, N. Zheng, S. Xu, Y. Qiao, Y. Liu, J. Li, and T. Zhang, "Kallaxdb: A table-less hash-based key-value store on storage hardware with built-in transparent compression," in *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, ser. DAMON'21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3465998.3466004

[18] Cisco, "Cisco Global Cloud Index: Forecast and Methodology, 2016–2021," *White Paper*, 2018.

[19] C. Dennl, D. Ziener, and J. Teich, "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)*, 2013, pp. 25–28.

[20] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1243–1254.

[21] J. Dittrich and A. Jindal, "Towards a One Size Fits All Database Architecture," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 195–198.

[22] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: opportunities and challenges," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1221–1230.

[23] J. Do, S. Sengupta, and S. Swanson, "Programmable solid-state storage in future cloud datacenters," *Communications of the ACM*, vol. 62, no. 6, pp. 54–62, 2019.

[24] ETHZ, "Enzian Systems," *http://enzian.systems/*, 2021.

[25] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, "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)*, 2019, p. 335.

[26] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on FPGAs: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 33–59, 2020.

[27] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database – An Architecture Overview," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 28–33, 2012.

[28] P. Francisco, "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics," *IBM Redbooks*, 2011.

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

[30] Google, "Cloud TPU," *https://cloud.google.com/tpu/*, 2017.

[31] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 153–165.

[32] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[33] M. Hassan, "Reduced latency DRAM for multi-core safety-critical real-time systems," *Real-Time Systems*, vol. 56, no. 2, pp. 171–206, 2020.

[34] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[35] Intel, Corp., "Intel's Stratix 10 FPGA: Supporting the smart and connected revolution," October 2016, accessed on 09.01.2020. [Online]. Available: https://tinyurl.com/IntelStratix2016

[36] Z. István, "The Glass Half Full: Using Programmable Hardware Accelerators in Analytics," *IEEE Data Engineering Bulletin*, vol. 42, no. 1, pp. 49–60, 2019.

[37] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A Flexible, High-Performance Key-Value SSD," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 373–384.

[38] K. Kara, J. Giceva, and G. Alonso, "FPGA-based Data Partitioning," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017, pp. 433–445.

[39] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki, "Adaptive Query Processing on RAW Data," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1119–1130, 2014.

[40] A. Kemper and T. Neumann, "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)*, 2011, pp. 195–206.

[41] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz, "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask," *Proceedings of the VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, 2018.

[42] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. S. Milojicic, and G. Alonso, "Farview: Disaggregated Memory with Operator Off-loading for Database Engines," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2022.

[43] C. Labs, "Vectorized Query Execution," *https://www.cockroachlabs.com/docs/stable/vectorized-execution.html*, 2019.

[44] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait, "Oracle Database In-Memory: A Dual Format In-Memory Database," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[45] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle TimesTen: An In-Memory Database for Enterprise Applications," *IEEE Data Engineering Bulletin*, vol. 36, no. 2, pp. 6–13, 2013.

[46] A. Lamb, M. Fuller, and R. Varadarajan, "The Vertica Analytic Database: C-Store 7 Years Later," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1790–1801, 2012.

[47] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving memory bank-level parallelism in the presence of prefetching," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 327–336. [Online]. Available: https://doi.org/10.1145/1669112.1669155

[48] G. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A Processing in Memory Taxonomy and a Case for Studying Fixed-function PIM," in *Proceedings of the Workshop on Near-Data Processing (WoNDP)*, 2013.

[49] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso, "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017, pp. 37–50.

[50] N. May, A. Böhm, and W. Lehner, "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)*, 2017, pp. 545–563.

[51] P. Menon, A. Pavlo, and T. C. Mowry, "Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 1–13, 2017.

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

[53] Microsoft, "Project Catapult," *https://www.microsoft.com/en-us/research/project/project-catapult/*, 2017.

[54] C. Mohan, "Hybrid Transaction and Analytics Processing (HTAP): State of the Art," in *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*, 2016.

[55] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Flexible Query Processor on FPGAs," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1310–1313, 2013.

[56] Oracle, "DAX," *https://blogs.oracle.com/linux/post/oracle-data-analytics-accelerator-dax-for-sparc*, 2021.

[57] F. Özcan, Y. Tian, and P. Tözün, "Hybrid Transactional/Analytical Processing: A Survey," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017, pp. 1771–1775.

[58] T. I. Papon and M. Athanassoulis, "A Parametric I/O Model for Modern Storage Devices," in *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2021.

[59] K. Park, Y.-S. Kee, J. M. Patel, J. Do, C. Park, and D. J. DeWitt, "Query Processing on Smart SSDs," *IEEE Data Engineering Bulletin*, vol. 37, no. 2, pp. 19–26, 2014.

[60] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "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)*, 2018, pp. 37–44.

[61] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A Case for Fractured Mirrors," *The VLDB Journal*, vol. 12, no. 2, pp. 89–101, 2003.

[62] A. Redshift, "Aqua (advanced query accelerator) for amazon redshift," 2021. [Online]. Available: https://aws.amazon.com/redshift/features/aqua/

[63] S. Roozkhosh, D. Hoornaert, J. H. Mun, T. I. Papon, A. Sanaullah, U. Drepper, R. Mancuso, and M. Athanassoulis, "Relational Memory: Native In-Memory Accesses on Rows and Columns," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2023, pp. 66–79.

[64] S. Roozkhosh and R. Mancuso, "The Potential of Programmable Logic in the Middle: Cache Bleaching," in *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 296–309.

[65] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim, "L-Store: A Real-time OLTP and OLAP System," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2018, pp. 540–551.

[66] B. Salami, G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sönmez, "AxleDB: A novel programmable query processing platform on FPGA," *Microprocessors and Microsystems*, vol. 51, pp. 142–164, 2017.

[67] A. Sanaullah, "Risc-v for fpgas: benefits and opportunities," *Red Hat Research Quarterly*, no. May, 2022.

[68] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User-Programmable SSD," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 67–80.

[69] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "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)*, 2015, pp. 267–280.

[70] N. Shamgunov, "The MemSQL In-Memory Database System," in *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, 2014.

[71] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso, "doppioDB: A hardware accelerated database," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[72] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "StRoM: smart remote memory," in *Proceedings of the EuroSys Conference (EuroSys)*, 2020, pp. 29:1—-29:16.

[73] M. Stonebraker and U. Cetintemel, ""One Size Fits All": An Idea Whose Time Has Come and Gone," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2005, pp. 2–11.

[74] X. Sun, C. J. Xue, J. Yu, T.-W. Kuo, and X. Liu, "Accelerating data filtering for database using FPGA," *Journal of Systems Architecture*, vol. 114, p. 101908, 2021.

[75] N. C. Thompson and S. Spanuth, "The decline of computers as a general purpose technology," *Communications of the ACM*, vol. 64, no. 3, pp. 64–72, 2021.

[76] TPC, "TPC-H benchmark," *http://www.tpc.org/tpch/*, 2021.

[77] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson, "SSD In-Storage Computing for Search Engines," *IEEE Transactions on Computers*, p. 1, 2016.

[78] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[79] L. Woods, Z. István, and G. Alonso, "Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

[80] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "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)*, 2014, pp. 255–268.

[81] ——, "The Q100 Database Processing Unit," *IEEE Micro*, vol. 35, no. 3, pp. 34–46, 2015.

[82] Xilinx, Inc., "Zynq UltraScale+ MPSoC - All Programmable Heterogeneous MPSoC," August 2016, accessed on 09.01.2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[83] M. Xue, Q. Xing, C. Feng, F. Yu, and Z.-G. Ma, "FPGA-Accelerated Hash Join Operation for Relational Databases," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67-II, no. 10, pp. 1919–1923, 2020.

[84] C. Zhang, R. Chen, and V. K. Prasanna, "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)*, 2016, pp. 148–155.

[85] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory (TIT)*, vol. 23, no. 3, pp. 337–343, 1977.

[86] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Super-Scalar RAM-CPU Cache Compression," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2006, p. 59.