POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database

Wei Cao[†]*, Yang Liu*[‡]*, Zhushi Cheng*[†]*, Ning Zheng*[‡]*, Wei Li*[†]*, Wenjie Wu*[†]*, Linqiang Ouyang*[‡]*,*
Peng Wang[†]*, Yijing Wang*[†]*, Ray Kuan*[‡]*, Zhenjun Liu*[†]*, Feng Zhu*[†]*, Tong Zhang*[‡]

[†] *Alibaba Group, Hang Zhou, Zhejiang, China*

[‡] *ScaleFlux Inc., San Jose, CA, USA*

Abstract

This paper reports the deployment of computational storage drives in Alibaba Cloud to enable cloud-native relational database cost-effectively support analytical workloads. With its compute-storage decoupled architecture, cloud-native relational database should pushdown data-intensive tasks (e.g., table scan) from front-end database nodes to back-end storage nodes in order to adequately support analytical workloads. This however makes it a challenge to maintain the cost effectiveness of storage nodes. The emerging computational storage opens a new opportunity to address this challenge: By replacing commodity SSDs with computational storage drives, storage nodes can leverage the in-storage computing power to much more efficiently perform table scans. Practical implementation of this simple idea is non-trivial and demands cohesive innovations across the software (i.e., database, filesystem and I/O) and hardware (i.e., computational storage drive) layers. This paper presents such a holistic implementation for Alibaba cloud-native relational database POLARDB. To the best of our knowledge, this is the first real-world deployment of cloud-native databases with computational storage drives ever reported in the open literature.

1 Introduction

Relational database is an essential building block in modern information technology infrastructure. Therefore, all the cloud vendors have invested significant efforts to grow their relational database service (RDS) business. Not surprisingly, some cloud vendors have developed their own cloud-native relational database systems, e.g., Amazon Aurora [28] and Alibaba POLARDB [9]. In order to achieve sufficient scalability and fault resilience, cloud-native relational databases naturally follow the design principle of decoupling compute from data storage [4, 17]. Meanwhile, they typically aim to be compatible with mainstream open-source relational databases (e.g., MySQL and PostgreSQL) and achieve high performance for OLTP (online transaction processing) workloads at a much lower cost than their on-premise counterparts.

It is highly desirable for cloud-native relational databases to adequately support analytical workloads. As pointed out by the authors of [28], because cloud-native relational databases decouple compute from data storage, the network bandwidth between database nodes and storage nodes becomes a scarce resource. This however does not match well to analytical workloads that involve intensive data access. To best serve OLTP workloads, cloud-native relational databases typically employ the row-store model (or the hybrid-row/column model [5]). This could make the network bandwidth an even bigger bottleneck for analytical workloads. In order to better serve analytical workloads, the almost only viable option is to off-load data-access-intensive tasks (in particular table scan) from database nodes to storage nodes. This concept is certainly not new and has been adopted by both proprietary database appliances (e.g., Oracle Exadata) and open-source databases (e.g., MySQL NDB Cluster). In spite of the simple concept, its practical implementation in the context of cloud-native databases is particularly non-trivial. On one hand, each storage node must be equipped with sufficient data processing power to handle table scan tasks. On the other hand, to maintain the cost effectiveness of cloud-native databases, we cannot significantly (or even modestly) increase the cost of storage nodes. By complementing CPUs with special-purpose hardware (e.g., GPU and FPGA), heterogeneous computing architecture appears to be an appealing option to address this data processing power vs. cost dilemma.

This work applies heterogeneous computing in POLARDB storage nodes to efficiently support table scan pushdown. The key idea is simple: Each POLARDB storage node off-loads and distributes table scan tasks from its CPU to its data storage devices. Under this framework, each data storage device becomes a *computational storage drive* [1] that can carry out table scan on the I/O path. Compared with off-loading table scan to a dedicated stand-alone computing device (e.g., FPGA/GPU-based PCIe card), distributing table scan across all the storage drives can minimize the data traffic across the storage/memory hierarchy and obviate data processing hotspot. This simple concept is not new and has been discussed

(e.g., see [11, 14]). However, its practically viable implementation and real-world deployment remain completely missing, at least in the open literature. This is mainly due to the difficulty of addressing two challenges: (1) how to practically support the table scan pushdown across the entire software hierarchy, and (2) how to implement low-cost computational storage drives with sufficient table scan processing capability.

Over the course of materializing this simple idea in the context of POLARDB on Alibaba Cloud, we developed a set of software/hardware techniques to cohesively address the two challenges. To reduce the product development cycle and meanwhile ensure cost effectiveness, computational storage drives use an FPGA-centric host-managed architecture. Inside each computational storage drive, a single mid-range low-cost Xilinx FPGA chip handles both flash memory control and table scan. With highly optimized software and hardware design, each computational storage drive can support high-throughput (i.e., over 2GB/s) table scan on compressed data and meanwhile achieve storage I/O performance comparable to leading-edge NVMe SSDs. We developed a variety of techniques that enable POLARDB storage nodes fully exploit the capability of computational storage drives. This paper presents these design techniques and elaborates on their implementation, and further presents evaluation results to demonstrate their effectiveness. Based on the TPC-H queries, we extracted six individual table scan tasks and ran these scan tasks on one storage node. Such node-level evaluation shows that the computational storage drives can largely reduce both scan latency and CPU utilization of the storage node. We further carried out system-level evaluations on a POLARDB cloud instance over 7 database nodes and 3 storage nodes. Results show that this solution can noticeably reduce the TPC-H query latency. To the best of our knowledge, this is the first application of emerging computational storage in production database ever reported in the open literature.

2 Background and Motivation

2.1 POLARDB: Basic Architecture

POLARDB is a new cloud-native OLTP database designed by Alibaba Cloud. Its design goals come from our cloud customers' real needs: large per-instance storage capacity (tens of TB), high TPS (transactions per second), high and scalable QoS and high availability. POLARDB provides enterprise-level cloud database services and is compatible with MySQL and PostgreSQL. Fig. 1 illustrates the compute-storage decoupled architecture of Alibaba POLARDB. Database computing nodes and storage nodes are connected through high-speed RDMA network. In each POLARDB instance, there is only one read/write database node that handles both the read and write requests, and the other database nodes handle only read requests. All the nodes in an instance, including read/write nodes and read-only nodes, are able to access the same copy of data on a storage node. To ensure the high availability, POLARDB uses the Parallel-Raft protocol to write three copies of data across the storage nodes [9].

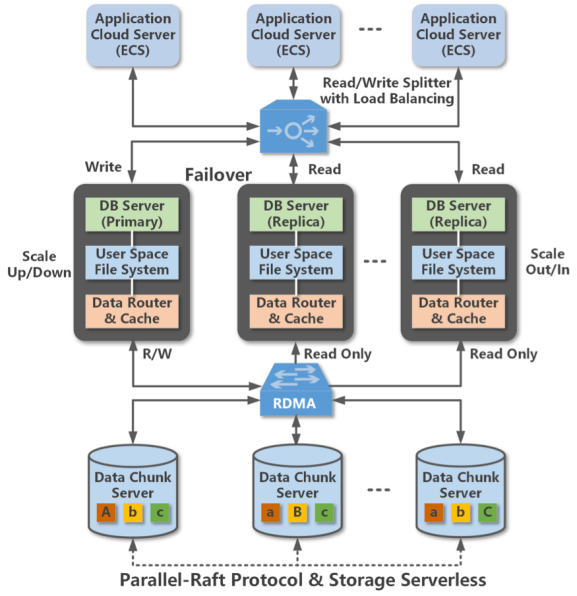

Figure 1: Illustration of POLARDB architecture.

2.2 POLARDB: Table Scan Pushdown

Off-loading table scan from database nodes to storage nodes is important for cloud-native relational database to effectively handle analytical workloads. This concept trades heavier data processing load on storage nodes for significantly reduced network traffic between database nodes and storage nodes. Moreover, since POLARDB employs the row-store model to better serve OLTP workloads, the column-oriented nature of table scan tends to demand even higher data processing power in storage nodes. Therefore, the key design issue is how to cost-effectively equip storage nodes with sufficient data processing power to handle the additional table scan tasks.

The most straightforward option is to simply scale up each storage node, which nevertheless is not practically desirable mainly due to the cost overhead. Table scan over row-store data does not fit well to modern CPU architecture and tends to largely under-utilize CPU hardware resources (e.g., cache memory, and SIMD processing resource) [2]. As a result, we have to more aggressively scale up the storage nodes to compensate for the inefficiency of CPU-based implementation. Hence, this straightforward option is economically unappealing and even unacceptable, especially as the classical CMOS technology scaling is quickly approaching its end [8].

An alternative is to complement storage node CPUs with special-purpose hardware (e.g., FPGA or GPU) that can carry out table scan with much better cost effectiveness. Under

this heterogeneous computing framework, the conventional practice uses a *centralized* heterogeneous architecture where the special-purpose hardware is implemented in the form of a single stand-alone FPGA/GPU-based PCIe card (e.g., see [24, 26, 29]). Nevertheless, this approach has several drawbacks for our targeted systems: (1) *High data traffic*: All the raw data in their row-store format must be fetched from the storage devices into the FPGA/GPU-based PCIe card. Due to the data-intensive nature of table scan, this leads to a very heavy data traffic over the PCIe/DRAM channels. The high data traffic can cause significant energy consumption overhead and inter-workload interference. (2) *Data processing hot-spot*: Each storage node contains a large number of NVMe SSDs, each of which can achieve multi-GB/s data read throughput. As a result, analytical processing workloads could trigger very high aggregated raw data access throughput that is far beyond the I/O bandwidth of one PCIe card. This could make the FPGA/GPU-based PCIe card become the system bottleneck.

The above discussion suggests that a *distributed* heterogeneous architecture is a better option. As illustrated in Fig. 2, by distributing table scans directly into each storage drive, we can eliminate the high data traffic over the PCIe/DRAM channels, and obviate data processing hot-spot in the system. This intuition directly motivated us to develop and deploy computational storage drives in POLARDB storage nodes.

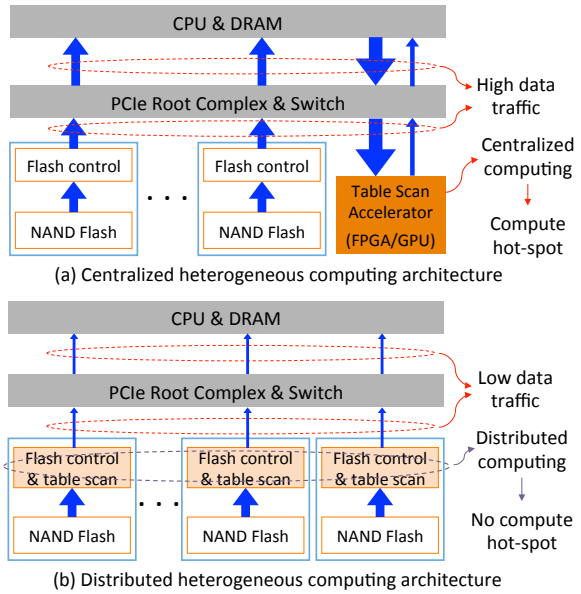

Figure 2: Illustration of (a) centralized heterogeneous computing architecture, and (b) distributed heterogeneous computing architecture.

2.3 Computational Storage Drive

Loosely speaking, any data storage device that can carry out data processing tasks beyond its core storage duty can be called a computational storage drive. The simple concept of empowering storage devices with additional computing capability can trace back to over 20 years ago [3, 21, 22]. Computational storage complements with CPU to form a heterogeneous computing system. Compared with its CPU-only counterpart, a heterogeneous computing system not surprisingly can achieve higher performance and/or energy efficiency for many applications, as demonstrated by prior research (e.g., see [10,11,15,16,18,23,27]). However, it is apparently subject to two cost overheads: (1) the hardware cost of implementing computational storage drives, and (2) the development cost on developing all the necessary hardware and software solutions to enable its real-world deployment. In spite of the over two decades of research, computational storage has not yet entered the mainstream market, arguably because of the absence of a practically justifiable benefit vs. cost trade-off.

To overcome the cost barrier, we chose an FPGA-based host-managed computational storage drive design strategy. This can reduce the development cost from two aspects: (1) We use a single FPGA to realize both flash memory control and computation (i.e., table scan in this work) inside computational storage drives. Compared with ASIC-based approach, the circuit-level programmability of FPGA can significantly reduce the computational storage drive development cycle and cost. (2) The computational storage drive is fully managed by the host for the functions such as address mapping, request scheduling, and garbage collection. Its host-management nature can facilitate integrating computational storage drive into existing software stack. It enables a high flexibility to devise and optimize the computational storage drive's API through which applications can utilize its configurable computation capability. Meanwhile, the host-managed computational storage drive natively integrates into the Linux I/O stack as a storage block device to serve normal I/O requests.

However, in return for its circuit-level programmability, FPGA is expensive (e.g., modern high-end FPGA chip could cost few thousand dollars), leading to a higher hardware cost of computational storage drive. Meanwhile, the objective of this work is to deploy computational storage drive to cost-effectively support table scan pushdown. Therefore, one key issue is how to minimize the hardware cost overhead while achieving sufficiently high storage I/O and table scan processing performance, which will be discussed in the next section.

3 Design and Implementation

As pointed out above, although applying computational storage to support table scan pushdown is a very simple concept and has been well discussed in the open literature, its real-world implementation and deployment has remained missing. Our first-hand experience of implementing this concept for POLARDB reveals that transferring this simple idea into real product faces the following two major challenges:

1. *Support table scan pushdown across the entire software hierarchy*: Table scan pushdown is initiated by the user-space POLARDB storage engine that accesses data by specifying the offsets in files, while table scan is physically served by computational storage drive that operates as a raw block device and manages data with LBA (logical block address). The entire storage I/O stack sits in between POLARDB storage engine and computational storage drive. Hence, we have to cohesively enhance/modify the entire software/driver stack in order to create a path in support of table scan pushdown.
2. *Implement low-cost computational storage drive*: As discussed above in Section 2.3, although the FPGA-based design approach can significantly reduce the development cost, FPGA tends to be expensive. Moreover, since FPGA typically operates at only 200∼300MHz (in contrast to 2∼4GHz CPU clock frequency), we have to employ a large degree of circuit-level implementation parallelism (hence more silicon resource) in order to achieve sufficiently high performance. Therefore, we must develop solutions to enable the use of low-cost FPGA chip in our implementation.

The remainder of this section presents a set of design techniques across the software and hardware stacks that can address the above two major challenges.

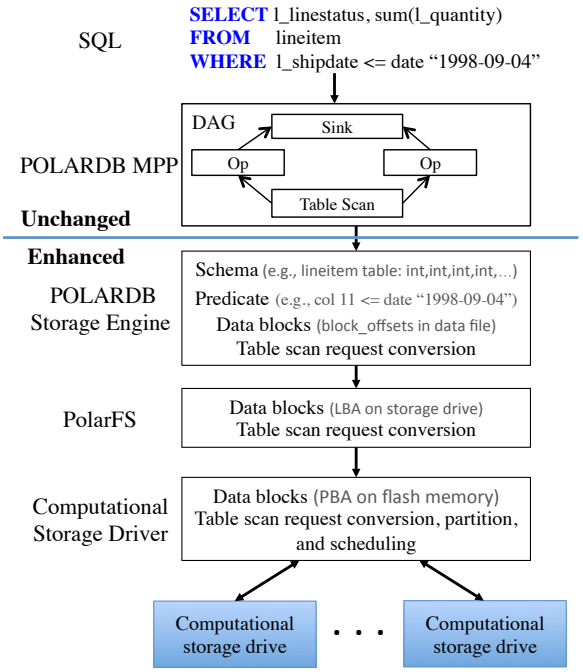

Figure 3: Illustration of the overall software stack.

3.1 Support Table Scan Pushdown Across the Entire Software Stack

To tackle the first challenge, we developed techniques to support the table scan pushdown across the entire software stack, as illustrated in Fig. 3. POLARDB database nodes incorporate a front-end analytical processing engine called *POLARDB MPP*. Being compatible with the MySQL protocol, this analytical processing engine can parse, optimize and rewrite SQL using the AST (abstract syntax tree) and a number of embedded optimization rules. It transforms each SQL query into a DAG (directed acyclic graph) execution plan consisting of operators and data flow topology. This analytical processing engine natively supports table scan pushdown to the underlying storage engine. Hence, we can keep the analytical processing engine intact in this work.

As illustrated in Fig. 3, in order to enable table scan pushdown, we have to appropriately enhance the entire storage stack underneath the analytical processing engine, including POLARDB storage engine, PolarFS (a distributed filesystem under POLARDB), and computational storage driver. In the following, we will elaborate on the implemented enhancements across these three layers.

3.1.1 Enhancement to POLARDB Storage Engine

POLARDB database storage engine follows the design principle of LSM-tree (log-structured merge-tree) [20]. Data in each table are organized into many files (typical file size is few tens of MBs), and each file contains a large number of blocks (typical block size ranges from 4KB to 32KB). In its original implementation, POLARDB storage engine serves the table scan requests using the CPUs on storage nodes. Hence, the underlying storage I/O stack is oblivious to the table scan pushdown. Since this work aims to utilize computational storage drives to process table scan, we have enhanced POLARDB storage engine so that it can pass table scan requests to the underlying filesystem PolarFS. As illustrated in Fig. 3, storage engine accesses data blocks in terms of offsets in files. Each table scan request contains: (1) the location (i.e., offsets in files) of the to-be-scanned data, (2) the schema of the table onto which the table scan is applied, and (3) the table scan conditions to be evaluated. Meanwhile, POLARDB storage engine allocates a memory buffer for storing data returned from computational storage drives, and each table scan request contains the location of this memory buffer.

As discussed later, the implemented computational storage drives do not support all the possible scan conditions (e.g., *LIKE* is not supported in current implementation). Hence, upon receiving table scan pushdown from the analytical processing engine, the enhanced storage engine first analyzes the scan conditions, and if necessary it extracts and passes a subset of the scan conditions that can be served by the computational storage drives. After receiving the data returned from the computational storage drives, the storage engine always checks the data against the complete table scan conditions. Moreover, to improve the overall system efficiency, we should

exploit the computational parallelism across multiple computational storage drives within each storage node. Therefore, POLARDB storage engine is able to issue multiple table scan requests concurrently to the underlying computational storage devices through PolarFS.

3.1.2 Enhancement to PolarFS

As described in [9], POLARDB is deployed on the distributed filesystem PolarFS that manages the data storage across all the storage nodes. Each computational storage drive can only perform table scan on its own data and meanwhile data are scanned in the unit of storage engine data blocks. Meanwhile, due to the use of block-level compression, variable-length compressed blocks are contiguously packed in each file (i.e., each compressed block is not 4KB-aligned). Therefore, PolarFS employs a coarse-grained data striping (4MB stripe size) across the computational storage drives in order to ensure most data blocks entirely reside on one computational storage drive. In the rare case of one compressed block locates across two drives, the system will use storage node CPU to handle the corresponding scan operation.

As discussed in Section 3.1.1, POLARDB storage engine specifies the location of to-be-scanned data in the form of offsets in files. The to-be-scanned data may span over multiple files and hence multiple computational storage drives. Meanwhile, computational storage drives can only locate data in the form of LBAs. Therefore, upon receiving each table scan request from POLARDB storage engine, PolarFS must appropriately convert this request before forwarding it to the computational storage driver. Accordingly, we have enhanced PolarFS from the following aspects: (1) Suppose the to-be-scanned data span over m computational drives, the enhanced PolarFS decomposes this request into m scan requests, each of which scans the data on one computational storage drive. (2) For each scan request, it converts the data location information into offsets in LBAs. As illustrated in Fig. 3, the enhanced PolarFS subsequently passes the m scan requests with converted LBA-based location information to the underlying computational storage driver.

3.1.3 Enhancement to Computational Storage Driver

As discussed above in Section 2.3, our computational storage drive is fully managed by a host-side driver in the kernel space. The driver exposes each computational storage drive as a block device. Upon receiving each table scan request from PolarFS, the driver carries out the following operations. It first analyzes the scan conditions, and if necessary re-arranges the scan conditions in order to better streamline the hardware-based scan processing and hence improve the throughput. For example, suppose the table contains 16 fields (i.e., f_1, f_2, $\cdots$, f_{16}), and the scan condition involves two comparisons, where the first one compares f_{10} and a constant, and the second one compares f_2 and f_5. Since hardware can pipeline the table record parsing, field selection, and comparison, if we re-arrange the scan condition by interchanging the position of the two comparisons, we can improve the hardware utilization efficiency and hence achieve higher processing throughput. The driver further converts the location information of the to-be-scanned data from the LBA domain into the physical block address (PBA) domain, where each PBA associates with a fixed location in NAND flash memory.

Moreover, the driver internally partitions each scan request into a number of (much) smaller scan sub-tasks, which can serve for two purposes: (1) A large scan task may occupy the flash memory bandwidth for a long time, which can cause other normal I/O request suffer from a longer latency. This problem can be mitigated by partitioning a large scan task into small sub-tasks and cohesively scheduling them with normal I/O requests. (2) By partitioning a large scan task into small sub-tasks, it helps to reduce the hardware resource usage for internal buffering and improve flash memory access parallelism. Moreover, storage device background operations, in particular garbage collection (GC), can severely interfere with table scan and hence cause significant latency penalty. Since all the flash management functions are handled by the host-side driver, we enhanced the driver so that it can cohesively schedule GC and table scan in order to minimize the GC-induced interference. In particular, in the case of heavy and bursty analytical processing workloads, the driver will adaptively reduce or even suspend the GC operation.

3.2 Reduce Hardware Implementation Cost

In order to tackle the challenge of computational storage drive implementation cost, the key is to maximize the FPGA hardware resource utilization efficiency. To achieve this objective, we further developed the following techniques across the software and hardware layers.

3.2.1 Hardware-Friendly Data Block Format

We first modified POLARDB storage engine data block format in order to facilitate the FPGA implementation of table scan. Table scan mainly involves various data comparison operations (e.g., $=$, $\geq$, $\leq$). In spite of the FPGA circuit-level programmability, it is difficult for FPGA to implement comparators that can efficiently support multiple different data types. In this work, we modified POLARDB storage engine so that it stores all the table data in the memory-comparable format, i.e., data can be compared using the function *memcmp*(). As a result, computational storage drives only need to implement a single type of comparator that can carry out the *memcmp*() function, regardless of the specific data types in different fields of a table. By enabling the implementation of type-oblivious comparators in FPGA, this can largely reduce the usage of FPGA resources for implementing table scan.

We further modified the storage engine data block structure in order to improve the hardware utilization efficiency. Fig. 4(a) illustrates the data block format being used in the original storage engine: One data block contains a number of sorted table entries, and ends with meta information (i.e., 1-byte data compression type and 4-byte CRC). Although such a block format can be easily handled by CPUs, it is not friendly to the hardware-based table scan in computational storage drives. We modified the data block format as illustrated in Fig. 4(b), where we add an additional block header including 1-byte block compression type, 4-byte number of key-value pairs, and 4-byte number of restart keys (note that restart key is used to facilitate key search in the presence of prefix compression). This modified block format is much more friendly to hardware-based table scan because: (1) Computational storage drive can decompress each block and check CRC without demanding POLARDB storage engine to pass the size information of each block. (2) By adding the "*# of keys*" and "*# of restarts*" fields at the beginning of each block, the hardware can more conveniently handle the restarts within each block and detect the end of each block. This is well suited to the sequential data processing flow of the hardware, and hence simplifies the FPGA-based hardware implementation.

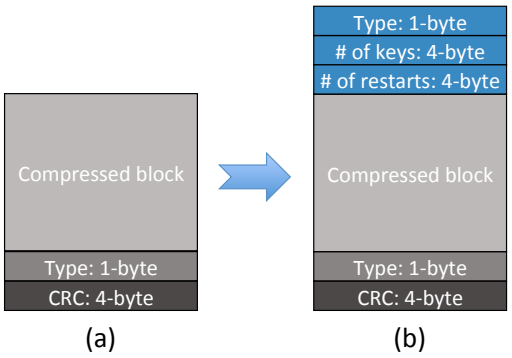

Figure 4: (a) Block structure in conventional practice, and (b) modified block structure to simplify hardware implementation of data scan.

3.2.2 FPGA Implementation

Fig. 5 shows the parallel and pipelined architecture of our FPGA implementation. To reduce the cost, we use a single mid-range FPGA chip for both flash memory control and table scan. The FPGA incorporates a powerful soft-decision LDPC (low-density parity-check) coding engine. This enables the use of low-cost 3D TLC (and QLC in the future) NAND flash memory, which helps to reduce the overall computational storage drive cost. We use a parallel and pipelined hardware architecture to improve the table scan processing throughput. As shown in Fig. 5, it contains two parallel data decompression engines and four data scan engines. Current implementation supports the Snappy decompression and following scan conditions: $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, NULL, and !NULL.

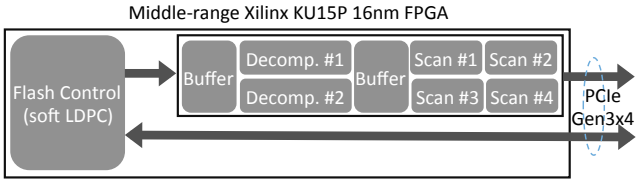

Figure 5: Parallel and pipelined FPGA implementation.

To further improve the hardware resource utilization efficiency, we applied a simple design technique described as follows. As pointed out above, all the fields are stored in the memory-comparable form, hence we only need to implement type-oblivious *memcmp* modules to evaluate each condition. Since the number of scan conditions varies among different table scan tasks, each scan engine employs a recursive architecture in order to maximize the FPGA resource utilization. Each scan engine contains one *memcmp* module and one *RE* (result evaluation) module. Let $P = \sum_{i=1}^{m}(\prod_{j=1}^{n_i} c_{i,j})$ denote the overall scan task, where each $c_{i,j}$ is one individual condition on one field. The symbols $\sum$ and $\prod$ represent the logic OR and AND operation, respectively. Using a single *memcmp* and *RE* module, we recursively evaluate the predicate with one condition $c_{i,j}$ at a time. The *RE* module checks whether the previous *memcmp* output (i.e., all the $c_{i,j}$'s that have been evaluated so far) is sufficient to determine the value of the result P. Once the value of P (i.e., either 1 or 0) can be determined, the scan engine can immediately finish the evaluation on current row, and start to work on another row. This recursive architecture can handle any arbitrary predicate with the optimal FPGA hardware resource utilization.

4 Evaluation

This section presents evaluation results to demonstrate the effectiveness of this deployed solution. The remainder of this section is organized as follows: Section 4.1 summarizes the experimental environment and basic storage performance of the computational storage drives. Section 4.2 evaluates and compares the table scan performance when using CPUs or computational storage devices to realize table scan. Section 4.3 presents the TPC-H evaluation results on a POLARDB instance in Alibaba Cloud, and Section 4.4 provides further concluding remarks.

4.1 Experimental Setup

In order to become practically viable products, besides providing in-storage computing capability, computational storage drives must have top-notch storage I/O performance (at least comparable with leading-edge commodity NVMe SSDs). The

storage performance of our computational storage drives is summarized as follows. Each drive uses 64-layer 3D TLC NAND flash memory chips. With PCIe Gen3×4 interface, each drive can sustain 2.2GB/s and 3.0GB/s sequential write and read throughput. Under 100% address span and fully triggered GC, each drive can achieve 160K and 590K random 4KB write and read IOPS, which are on par with the latest enterprise-grade NVMe SSDs. Each computational storage drive hosts a single mid-range Xilinx UltraScale+ KU15p FPGA chip that handles both flash memory control and computation. To maximize the error correction strength, each drive supports soft-decision LDPC code decoding with beyond-3GB/s decoding throughput. The performance evaluation is carried out on a POLARDB instance (with seven database nodes and three storage nodes) in Alibaba Cloud.

4.2 Table Scan Performance Evaluation

The FPGA inside each computational storage drive incorporates two Snappy decompression engines and four data scan engines. The decompression throughput varies with the data compressibility. Under compression ratio of 60% and 30%, the two decompression engines total can achieve 2.3GB/s and 2.8GB/s decompression throughput, respectively. The data scan engines also have variable throughput that depend on several runtime parameters, e.g., the size of each row in the table, table schema, and scan conditions.

We uses the *LINEITEM* table defined in TPC-H benchmark as a test vehicle to evaluate the effectiveness of moving table scan to computational storage drives. The *LINEITEM* table contains total 16 columns mixed with data types of identifier, integer, decimal, fixed-length and variable-length strings. To cover a wide range of processing complexity, we chose the following six table scan tasks (extracted from different TPC-H queries) to carry out evaluations on one storage node:

TS-1: *Select* L_PARTKEY, L_EXTENDEDPRICE, L_DISCOUNT
from LINEITEM
where L_SHIPDATE ≥ "1994-06-01" and L_SHIPDATE < "1994-07-01"

TS-2: *Select* L_PARTKEY, L_SUPPKEY, L_QUANTITY
from LINEITEM
where L_SHIPDATE ≥ "1993-01-01" and L_SHIPDATE < "1994-01-01"

TS-3: *Select* L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE, L_DISCOUNT, L_SHIPDATE
from LINEITEM
where L_SHIPDATE ≥ "1995-01-01" and L_SHIPDATE ≤ "1996-12-31"

TS-4: *Select* L_ORDERKEY, L_EXTENDEDPRICE, L_DISCOUNT
from LINEITEM
where L_SHIPDATE ≤ "1995-03-12"

TS-5: *Select* L_ORDERKEY
from LINEITEM
where L_COMMITDATE < L_RECEIPTDATE

TS-6: *Select* L_PARTKEY, L_SUPPKEY, L_QUANTITY
from LINEITEM

For the above six scan tasks, the data selectivity in terms of table entries is 1.25%, 15.17%, 30.34%, 54.04%, 63.22%, and 100.00%, respectively. We set the raw data compression ratio as 0.5 when generating the LINEITEM table, and use the Snappy compression library to compress each data block. For each table scan task, we measured the scan latency and PCIe data traffic when turning on and off the table scan pushdown. When we turn off the table scan pushdown, storage node treats each computational storage drive as a normal SSD and relies on CPU to carry out the table scan processing.

Fig. 6 shows the measured scan latency and CPU utilization, where each data point is obtained by averaging the results of 10 independent runs. As discussed above, each computational storage drive contains four hardware data scan engines. Hence, the storage node runs the scan tasks under two hardware configurations: (a) one computational storage drive with 4 CPU threads, and (b) two computational storage drives with 8 CPU threads. The notation *CPU-based Scan* and *CSD-based Scan* correspond to the cases when storage nodes use its CPU and computational storage drives to carry out table scan processing, respectively. As shown in Fig. 6, under each hardware configuration, we studied four cases: (1) *CPU-based scan* without data compression, (2) *CSD-based scan* without data compression, (3) *CPU-based scan* with Snappy compression, and (4) *CSD-based scan* with Snappy compression.

The results clearly show that, compared with *CPU-based scan*, its CSD-based counterpart can *simultaneously* reduce the scan latency and CPU utilization. For example, when we run the scan task TS-1 (with Snappy compression) on two drives with 8 threads, *CSD-based scan* can reduce the latency from 55s to 39s and meanwhile reduce the CPU utilization from 514% to 140%. Compared with other scan tasks, TS-6 can least benefit from CSD-based scan because its very simple scan condition largely under-utilizes the hardware resource in computational storage drives. Even for TS-6 (with Snappy compression), when using two drives with 8 threads, *CSD-based scan* can reduce the latency from 65s to 53s and meanwhile reduce the CPU utilization from 558% to 374%. Fig. 6 also shows that, although the CPU utilization of CPU-based scan remain relatively constant across all the six scan tasks, the CPU utilization of *CSD-based scan* noticeably increases as the data selectivity becomes larger. For example, TS-1 (with the selectivity of 1.25%) and TS-2 (with the selectivity of 15.17%) have less CPU utilization than others. This can be explained as follows: In the case of *CSD-based scan*, the CPU workload is proportional to the data selectivity. The smaller the data selectivity is, the less amount of data are transferred to and processed by the host CPU. In contrast, in the case of *CPU-based scan*, regardless of the data selectivity, host CPU has to fetch and process all the data from drives. The

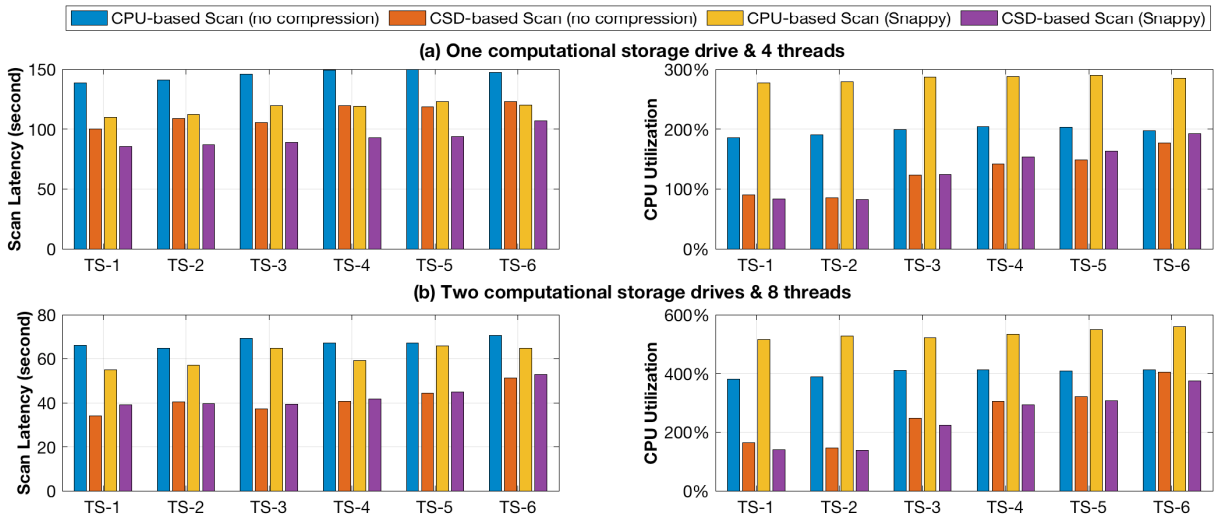

Figure 6: Measured scan latency and CPU utilization when the storage node runs the scan tasks on (a) one computational storage drive with 4 CPU threads, and (b) two computational storage drives with 8 CPU threads.

results also show that the effectiveness of *CSD-based scan* can readily scale with the number of computational storage drives. Finally, the results reveal that light-weight compression (i.e., Snappy in this study) can noticeably improve the performance of *CPU-based scan* at the cost of CPU utilization. In comparison, *CSD-based scan* is relatively insensitive to the use of compression.

To further reveal the benefit of using computational storage table scan pushdown to reduce data movement across the storage and memory hierarchy, Fig. 7(a) shows the measured volume of data being transferred from computational storage drives to host DRAM, and Fig. 7(b) shows the measured total host memory data transfer volume. The results show that

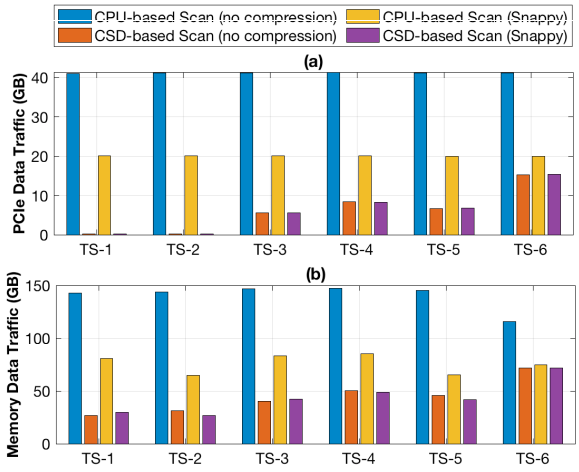

Figure 7: (a) PCIe data traffic and (b) memory data traffic inside the storage node.

CSD-based scan can significantly reduce the data transfer volume across the storage and memory hierarchy. The benefit improves as the data selectivity becomes smaller. For example, in the case of scan task TS-1 (with the selectivity of 1.25%) , *CSD-based scan* can almost eliminate the PCIe data transfer traffic, and reduce the host memory data traffic by 5× (without compression) and 3× (with compression). The results also show that compression can very effectively reduce data traffic volume across the storage and memory hierarchy.

4.3 System-level Evaluation

We further ran TPC-H analytical workload benchmark on a POLARDB cloud instance with 32 SQL-engine containers distributed on 7 database nodes and 3 back-end storage nodes. Each storage node hosts 12 computational storage drives, and each drive has a capacity of 3.7TB. We considered the following three different scenarios:

1. *No pushdown*: In this baseline scenario, database nodes do not push the table scan down to storage nodes. As a result, storage nodes have to transfer all the data to database nodes for table scan.
2. *CPU-based pushdown*: We enable the table scan pushdown from database nodes to storage nodes, and the CPUs on the storage nodes are responsible for carrying out table scan.
3. *CSD-based pushdown*: We enable the table scan pushdown from database nodes to storage nodes, and the computational storage drives on the storage nodes are responsible for carrying out table scan.

For each one out of the total 22 TPC-H queries, we measured the POLARDB performance by splitting data into partitions and submitting n scan requests in parallel to the back-end storage cluster. In this study, we considered three different

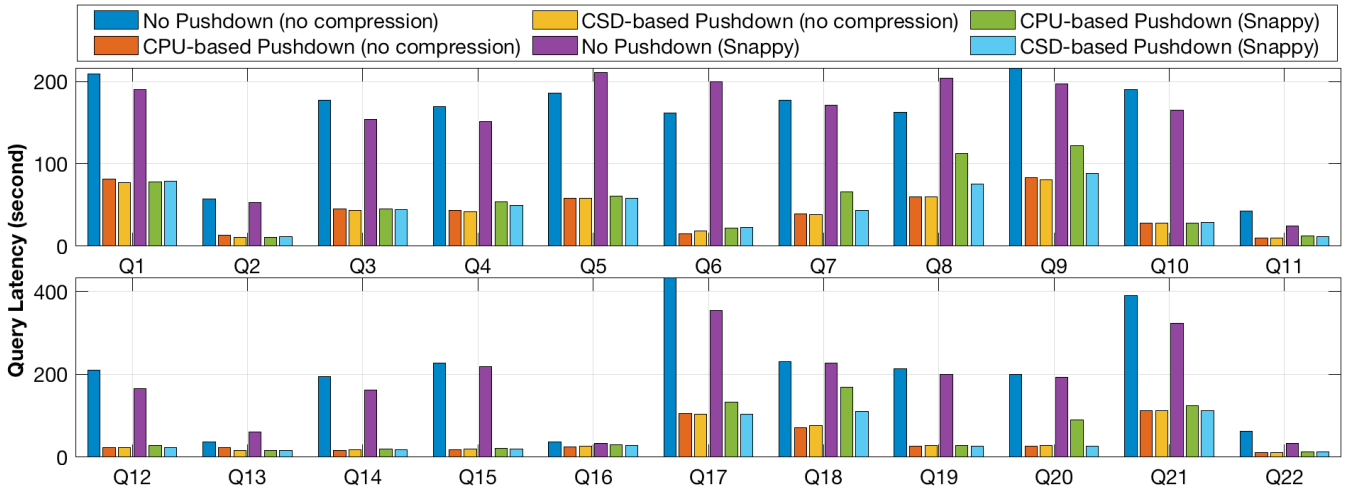

Figure 8: Measured TPC-H query latency under 32 parallel requests.

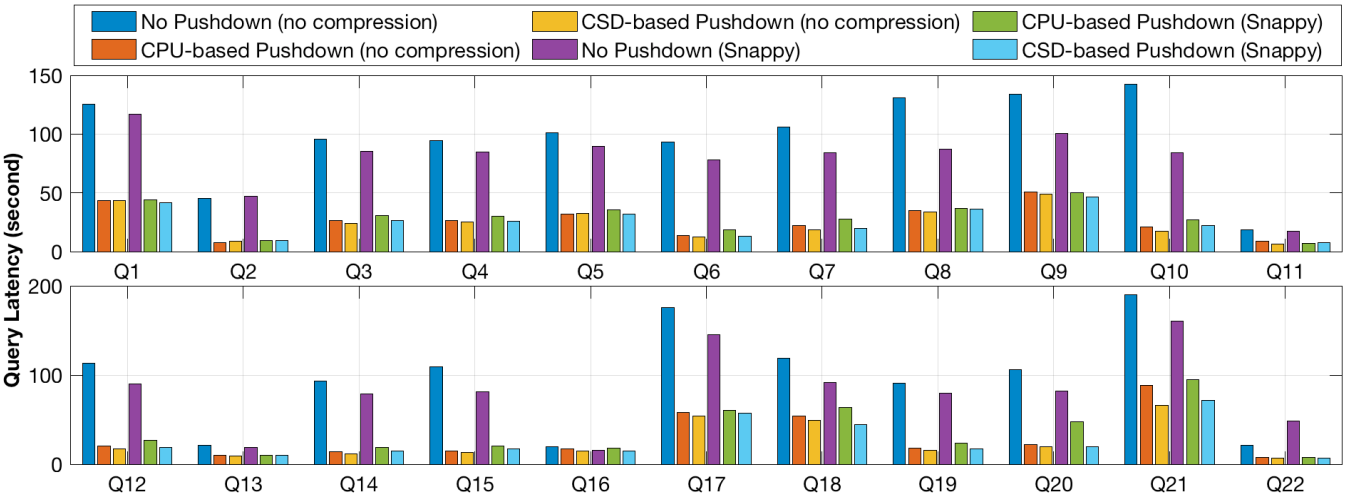

Figure 9: Measured TPC-H query latency under 64 parallel requests.

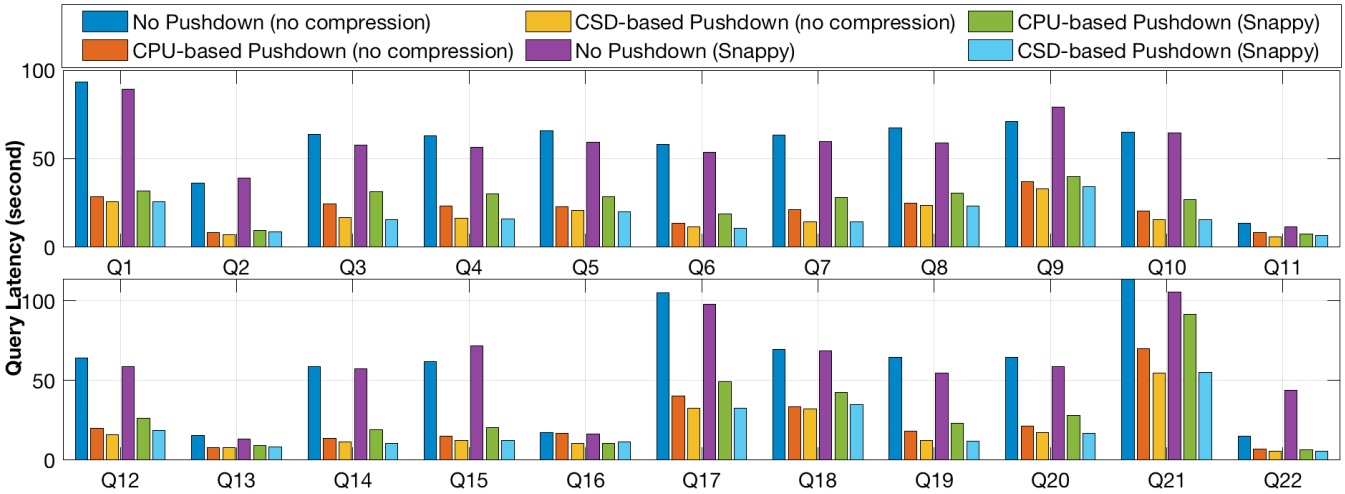

Figure 10: Measured TPC-H query latency under 128 parallel requests.

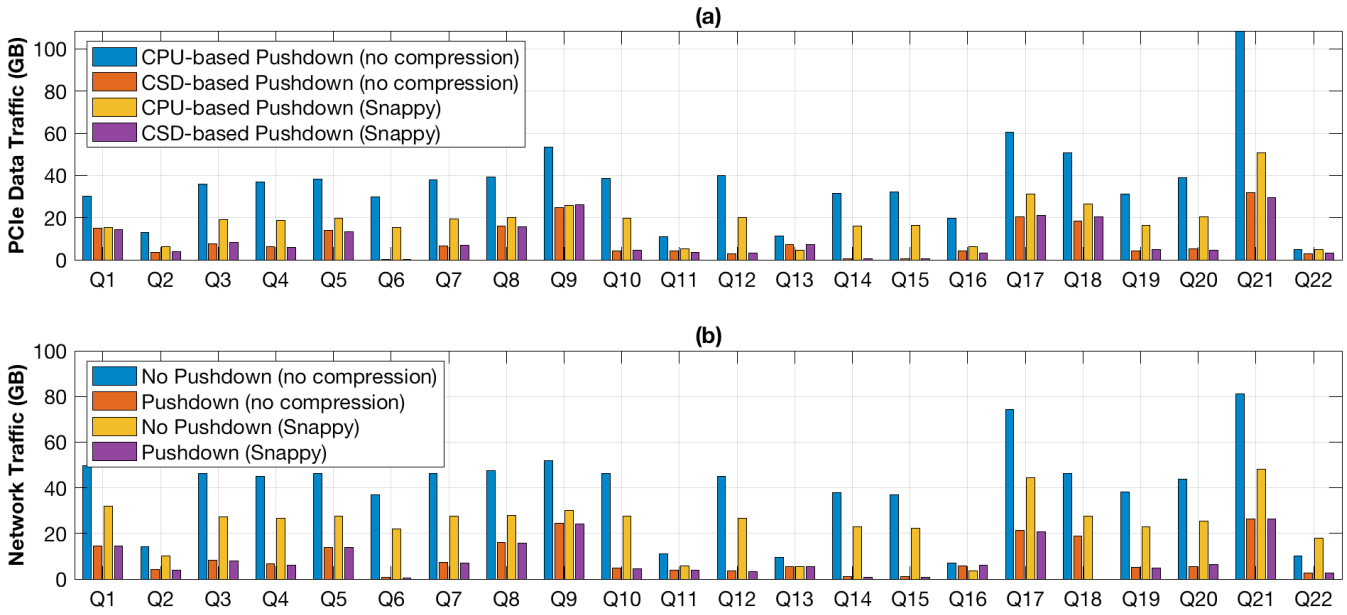

Figure 11: (a) PCIe data traffic inside storage nodes and (b) network data traffic in the POLARDB cluster.

values of n: 32, 64, and 128. Fig. 8, Fig. 9, and Fig. 10 show the measured latency of all the 22 TPC-H queries under 32, 64, and 128 parallel requests, respectively. Each evaluation point is obtained by averaging the results of 5 independent runs. The results clearly show the significant benefit of migrating table scan operations from database nodes to storage nodes, which can be intuitively justified given the compute-storage decoupled architecture of POLARDB. The results show that, as the number of requests increases, *CSD-based pushdown* on average can more noticeably outperform *CPU-based pushdown* in terms of scan latency. For example, in the case of 32 parallel requests (with Snappy compression), when switching from *CPU-based pushdown* to *CSD-based pushdown*, only 4 queries experience more than 30% latency reduction. In contrast, in the case of 128 parallel requests (with Snappy compression), when switching from *CPU-based pushdown* to *CSD-based pushdown*, 11 queries experience more than 30% latency reduction, where the maximum latency reduction is 50% for Q7. This is because, as the number of parallel requests increases, storage nodes will have more parallel table scan tasks to better utilize the hardware resource in the computational storage drives. Moreover, the results show that the benefit of *CSD-based pushdown* tends to improve when table data are compressed by Snappy. This can be explained as follows: When table data are compressed, *CPU-based pushdown* will consume more CPU resource in order to handle both data decompression and query processing. Hence a larger number of parallel requests will more likely make *CPU-based pushdown* CPU-bound. In contrast, *CSD-based pushdown* can readily leverage the hardware decompression engines in computational storage drives.

The results also show that *CPU-based pushdown* may even slightly outperform *CSD-based pushdown* in few cases under 32 or 64 requests (e.g., Q10 with 32 requests). This is most likely caused by the sub-optimal behavior of table scan pushdown scheduling, which leads to significant under-utilization of the hardware resource in the computational storage drives. Our future work will focus on improving the quality of table scan pushdown scheduling in order to avoid significant hardware resource under-utilization. Finally, Fig. 11 shows the measured total volume of PCIe data traffic inside storage nodes and total volume of network data traffic between database nodes and storage nodes. When switching from CPU-based pushdown to CSD-based pushdown, 7 TPC-H queries (with Snappy compression) experience more than 50% reduction on the PCIe data traffic volume, where the maximum PCIe data traffic volume reduction is 97% for Q6 followed by 94% for Q14. By moving table scan from database nodes to storage nodes, 12 TPC-H queries (with Snappy compression) experience more than 70% reduction on the total network data traffic volume. The above results clearly demonstrate the significant reduction in data traffic and scan latency of table scan pushdown in cloud-native database.

4.4 Summary

In-storage computing is a very simple concept and has been well discussed in the research community. Nevertheless, its practical implementation and deployment in real systems has remained elusive. Meanwhile, it is not uncommon that significant gain at the component level does not translate to noticeable benefit at the system level. Hence, commercializing the

simple idea of in-storage computing goes far beyond implementing a storage device that can do certain computation, and demands cohesive innovations across software and hardware hierarchy. Targeting at bringing in-storage table scan to cloud-native database systems, we have developed holistic solutions across the storage engine, filesystem, driver, and hardware stack. The component-level evaluation results in Section 4.2 show that our implemented computational storage drive can achieve high-throughput in-storage table scan, leading to significant reduction on host CPU usage and storage-to-memory data movement. The system-level evaluation results in Section 4.3 show that our holistic solution indeed can carry the component-level gain to the system level. The system-level evaluation also confirms the critical importance of realizing table scan pushdown from database nodes to storage nodes.

5 Related Work

Prior work has well studied the promise of accelerating databases using special-purpose hardware (in particular FPGA and GPU) to complement with CPUs. Many prior efforts focused on off-loading the table scan in analytical processing to dedicated accelerators (typically in the form of PCIe cards) built with either FPGA [24,26,29] or GPU [7,25]. Beyond table scan, prior work also investigated the potential of off-loading more complicated query processing kernels [12, 19, 30]. Nevertheless, in spite of extensive prior efforts and impressive performance benefits being demonstrated over the years, IBM/Netezza [24] appears to be the only known commercially successful product on mainstream markets. It off-loads data compression and table scan into dedicated FPGA-based PCIe cards in IBM PureData Systems. Beyond using stand-alone accelerators to complement CPUs, Oracle even integrated special-purpose analytics acceleration units into its own SPARC CPU [6], which however apparently suffers from a very high development cost and has been discontinued by Oracle.

The emerging computational storage enables new opportunities to implement heterogeneous computing platforms for databases. The authors of [13] studied the design of computational storage drives that support key-value store. Prior work [11, 14] focused on leveraging computational storage drives to realize in-storage table scan. Although prior work [11, 14] share the same basic concept as this work, there are several distinct differences: (1) This work presents a holistic system solution in the context of cloud-native relational database, and demonstrates its effectiveness in real production environment. In comparison, prior work [11] ran synthetic queries inside one computational storage drive without integration with databases and system I/O stack. Prior work [14] implemented a prototype based on a modified MySQL running on a single server. It did not consider the integration with a database system with compute-storage decoupled architecture, and did not consider the use of multiple computational storage drives in one server. (2) The basic storage I/O performance metrics (i.e., sequential throughput and IOPS) of the computational storage drives being used in prior work are much worse than that of leading-edge commodity NVMe SSDs. As a result, the systems in prior work tend to be much more I/O-bound and hence more easily benefit from in-storage table scan. The benefits shown in prior work may largely diminish when being compared with systems that deploy leading-edge commodity NVMe SSDs. (3) Both prior work [11, 14] use embedded processors within SSD controllers to carry out the data processing, which however cannot match the multi-GB/s intra-SSD NAND flash memory access bandwidth and hence cannot achieve high-throughput predicate evaluation. (4) Data compression is widely used in databases to reduce the storage bit cost. As a result, computational storage drives must carry out data decompression in order to support predicate evaluation on the data read path. However, prior work [11, 14] did not consider the implementation of data decompression.

6 Conclusions

This paper reports a cohesive cross-software/hardware implementation that enabled Alibaba cloud-native relational database POLARDB to effectively support analytical workloads. The basic design concept is to dispatch the costly table scan operations in analytical processing from CPU into computational storage drives. Being well aligned with current industrial trend towards heterogeneous computing, the key idea is very simple and can trace back to over two decades ago. Nevertheless, it is non-trivial to practically materialize this simple idea with justifiable benefit vs. cost trade-off in the real world. Under the framework of Alibaba POLARDB, this work developed a set of design solutions across the entire software and hardware stacks to practically implement this simple idea in production cloud database environment. Experimental results on a POLARDB cloud instance over 7 database nodes and 3 storage nodes show that our implementation can achieve more than 30% latency reduction for 12 out of the total 22 TPC-H queries. Meanwhile, our implementation can reduce more than 50% storage-to-memory data movement volume for 12 TPC-H queries. It is our hope that this work will inspire much more research and development efforts to investigate how future cloud infrastructure can leverage the emerging computational storage drives.

References

[1] *SNIA Technical Work Group on Computational Storage*. https://www.snia.org/computational.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. Row-stores: How different are they really? In

Proceedings of the International Conference on Management of Data (SIGMOD), pages 967–980, 2008.

[3] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, 1998.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[5] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, Nov. 2002.

[6] K. Aingaran, S. Jairath, and D. Lutz. Software in silicon in the Oracle SPARC M7 processor. In *IEEE Hot Chips Symposium (HCS)*, pages 1–31, 2016.

[7] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[8] M. T. Bohr and I. A. Young. CMOS scaling trends and beyond. *IEEE Micro*, 37(6):20–29, November 2017.

[9] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, Aug. 2018.

[10] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: A case for intelligent SSDs. In *Proc. of the International ACM Conference on Supercomputing*, pages 91–102, 2013.

[11] 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 (SIGMOD)*, pages 1221–1230, 2013.

[12] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. FPGA-based multithreading for in-memory hash joins. In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, 2015.

[13] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, Aug. 2017.

[14] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong. YourSQL: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, Aug. 2016.

[15] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. BlueDBM: An appliance for big data analytics. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2015.

[16] Y. Kang, Y.-S. Kee, E. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *Proc. of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2013.

[17] J. J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[18] D. Li, F. Wu, Y. Weng, Q. Yang, and C. Xie. HODS: Hardware object deserialization inside SSD storage. In *Proc. of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 157–164, 2018.

[19] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *Proc. VLDB Endow.*, 6(12):1310–1313, Aug. 2013.

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[21] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, Mar 1997.

[22] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pages 62–73, 1998.

[23] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable SSD. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 67–80, 2014.

[24] M. Singh and B. Leonhardi. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 385–386, 2011.

[25] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:8, 2013.

[26] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420, 2012.

[27] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. Reducing data movement costs using energy efficient, active computation on ssd. In *Proc. of the USENIX Conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[28] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[29] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[30] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *proc. of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 107–118, 2012.