Deploying Computational Storage for HTAP DBMSs Takes More Than Just Computation Offloading

Kitaek Lee*
Hanyang University
ktlee20@hanyang.ac.kr

Insoon Jo*†
Hanyang University
insoonjo@hanyang.ac.kr

Jaechan Ahn*
Hanyang University
jaechanahn@hanyang.ac.kr

Hyuk Lee
Samsung Electronics
hyuk17.lee@samsung.com

Hwang Lee
Samsung Electronics
h3.lee@samsung.com

Woong Sul
Hanyang University
woongsul@hanyang.ac.kr

Hyungsoo Jung‡
Hanyang University
hyungsoo.jung@hanyang.ac.kr

ABSTRACT

Hybrid transactional/analytical processing (HTAP) would overload database systems. To alleviate performance interference between transactions and analytics, recent research pursues the potential of in-storage processing (ISP) using commodity computational storage devices (CSDs). However, in-storage query processing faces technical challenges in HTAP environments. Continuously updated data versions pose two hurdles: (1) data items keep changing, and (2) finding visible data versions incurs excessive data access in CSDs. Such access patterns dominate the cost of query processing, which may hinder the active deployment of CSDs.

This paper addresses the core issues by proposing an ***a**nalyt**i**c offloa**d** **e**ngine* (AIDE) that transforms engine-specific query execution logic into vendor-neutral computation through a canonical interface. At the core of AIDE are the *canonical representation* of vendor-specific data and the separate management of data locators. It enables any CSD to execute vendor-neutral operations on canonical tuples with separate indexes, regardless of host databases. To eliminate excessive data access, we *prescreen* the indexes before offloading; thus, host-side prescreening can obviate the need for running costly version searching in CSDs and boost analytics. We implemented our prototype for PostgreSQL and MyRocks, demonstrating that AIDE supports efficient ISP for two databases using the same FPGA logic. Evaluation results show that AIDE improves query latency up to 42× on PostgreSQL and 34× on MyRocks.

PVLDB Reference Format:
Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. Deploying Computational Storage for HTAP DBMSs Takes More Than Just Computation Offloading. PVLDB, 16(6): 1480 - 1493, 2023.
doi:10.14778/3583140.3583161

PVLDB Artifact Availability:
Artifacts are available at https://github.com/hyu-scslab/SmartSSD.

*These authors share co-first authorship
†Work done while at Samsung
‡Corresponding author and principal investigator

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583161

1 INTRODUCTION

Larger-than-memory analytics have been a significant matter in databases since it consumes substantial storage and computing power. The growing demand for HTAP deepens the situation because resource contention between transactions and analytics would rise steeply and badly impact the overall performance of databases. In particular, unpredictable data analytics often overburden database servers, leading performance metrics—transaction throughput, query latency, and storage overhead—to fluctuate severely. These significant challenges led to a busy research area, proposing system designs, such as extract-transform-load (ETL)-based systems [5, 21, 28, 30, 32, 39, 41, 58], for better performance isolation. As a promising way of isolating heterogeneous workloads and alleviating database loads, some research has explored the potential of computational storage to offload all or part of query execution plans to storage capable of in-storage (or in-situ) processing (ISP).

We have seen many proposals [10, 24, 54] in ISP for OLAP workloads, with a recent study [47] exploring ISP for HTAP. One of the difficulties faced by emerging research is continuously updated data. To overcome the problem, Tobias et al. [47] proposed an update-aware near-data processing architecture that accumulates update deltas and shares them with computational storage. Despite such improvements, computational storage devices are hard-wired to host databases since CSDs should understand database storage layouts and semantics to execute version traversal logic, which causes significant overhead in accessing required data versions [26, 27]. If we ever offload version searching logic to CSDs, it may undermine the rationale for fast analytics due to random, redundant data access in computational storage. So, *forestalling excessive data access is crucial in designing ISP frameworks for HTAP databases.*

We address the challenges by proposing an ***a**nalyt**i**c offloa**d** **e**ngine* (AIDE), an intermediate layer aiding host databases for offloading vendor-specific analytics to computational storage. We also explore potential sweet spots of ISP in HTAP. To this end, AIDE relies on two features: (1) *vendor-neutral computation* with *version indexes* and (2) *prescreening of version indexes*. For the former, AIDE transforms vendor-specific query execution into vendor-neutral one and provides CSDs with data version indexes holding *canonical tuple identifiers* (CTIDs) for accessing target data in a vendor-neutral way. By doing this, the "vendor-neutrality" truly enables FPGA kernels in CSD to be reusable in any engine without costly reengineering work. Vendor-neutral computation in AIDE

supports a limited set of primitive operations; for our prototype system, we support operations, such as hashing and exact matching on primitive types, offered by commodity CSDs.

Next, one must deal with version searching if ISP runs for MVCC databases. MVCC systems first find visible record versions satisfying query snapshots and execute query operators. In HTAP, transactions continuously update target tables, producing version streams for tuples and increasing the cost of finding visible data versions. Prior studies [26, 27] have identified increased overhead for version lookup in HTAP as one of the main culprits responsible for performance problems in MVCC systems. Hence, we exclude the version searching logic from offloaded computation to uphold vendor neutrality and unburden devices, which questions how to provide visible data versions to computational storage. AIDE addresses this issue by prescreening version indexes to filter invisible data versions before passing the indexes to computational storage. Since it only provides visible CTIDs to storage, ISP can avert random, redundant access to data files, thus accelerating some queries even under update-heavy workloads.

To validate our claims, we implemented AIDE for two full-fledged databases having completely different storage layouts: PostgreSQL and MyRocks. We also implemented the vendor-neutral execution logic in commodity computational storage: Samsung SC1733 SSD [53] having 4 TiB capacity with Xilinx field-programmable gate arrays (FPGAs) embedded. We use the same FPGA logic for two database systems, while the host-side AIDE component is hardwired to its host engine to transform engine-specific semantics into vendor-neutral ones. Experimental evaluation demonstrates that AIDE helps databases execute query offloading to CSDs, reduce resource contention, and boost analytics (i.e., 42× on PostgreSQL and 34× for MyRocks). To the best of our knowledge, AIDE is the first work enabling ISP in HTAP through vendor-neutral computation.

In summary, we make the following contributions:

- We show that offloading query analytics naively in HTAP is inefficient due to excessive data access that is detrimental to databases pursuing an efficient offloading framework.
- We address the technical challenges by introducing an analytic offload engine that offloads vendor-neutral computation to CSDs and uses the prescreening technique.
- Vendor-neutral computation allows CSDs to run analytics without vendor-specific semantics. Prescreening eliminates excessive data access in CSDs and contributes to excluding version-searching logic from CSDs.
- Evaluation shows that AIDE can aid HTAP DBMSs to offload query operations to ease resource contention and boost some suitable queries dramatically but also exhibit limitations in handling other queries in CSDs.

2 BACKGROUND AND RELATED WORK

2.1 Background

The past decade has witnessed a plethora of research exploring the benefits of offloading computation from host computer systems to computational devices (e.g., computational storage [22, 31, 45, 46, 48, 49, 56] and programmable network interface cards [29, 36, 40, 57]). Among them is in-storage processing of query executions that unloads data analytics on embedded processors in storage devices to gain performance benefits, mainly due to near-data accessibility. These studies implement either overall database operations, or a subset of operations in favor of the in-storage processing approach. In essence, computation offloading comprises two components: operations and data. Operations must be clearly defined for programming easiness, and data accessed by operations should be readily accessible. From this perspective, one can transform database operations into vendor-neutral formats, but data requires specific access methods to understand vendor-defined data layouts. Since modern databases tightly integrate access methods with relational operations, query offloading may significantly increase development costs when one implements in-storage processing of database operations, thus making such implementations difficult to reuse.

Table 1: Literature review on in-storage query processing

	Vendor-neutral	Supported Ops.	Result size	Workload
YourSQL [24]	No	Filter	Unlimited	TPC-H
Ibex [52]	No	Filter, Agg.	Unlimited	TPC-H
POLARDB [10]	No	Filter	Unlimited	TPC-H
AQUOMAN [54]	No	Filter, Agg., Join, Sort	Limited	TPC-H
Tobias et al. [47]	No	Filter, Agg., Join	Unlimited	TPC-CH
AIDE	Yes	Filter, Join	Unlimited	TPC-CH

Commercial or open-source databases [6, 7, 14, 16, 18, 19, 23, 25, 33, 34] relying on multiversion concurrency control (MVCC) [8, 9, 35, 37] maintain multiple record versions to provide queries with point-in-time consistency. Under HTAP loads, MVCC databases keep producing new data versions and increase the search space for version lookup. Version searching on this increased space delays query analytics substantially and may significantly devalue the performance benefits of in-storage processing. Moreover, DBMS vendors differ in what information to store for a version when managing multi-versioned data and how to access a visible version. The wide variety of version management makes version retrieval rely heavily on vendor-specific data layouts and makes a vendor-neutral form of computation offloading challenging. Recent studies [26, 27] have reported results of comprehensive analyses on performance issues arising in MVCC databases under HTAP workloads.

2.2 Related Work

In attempts to adopt ISP in the database landscape, prior proposals have used embedded processors [15, 24, 38], special-purpose hardware [42, 51], GPUs [17, 20], or FPGAs [10, 47, 50, 52, 55] to accelerate database analytics. In particular, FPGAs have attracted attention due to their user programmability at a relatively low cost, and vendors often combine storage devices with FPGA hardware. Since the practical use of ISP demands the instruction-level regularity and parallelism of the hardware, diverse characteristics of internal workflow or unpredictable delays may not be suitable to query offloading. From this perspective, most studies have used static datasets, which can simplify hardware logic in accessing data. For instance, FCAccel [50] modifies the data layout for operations in hardware, and others [10, 55] use metadata to access tuples. However, MVCC databases undergo unpredictable access latencies for fetching visible data versions, unfavorable to query offloading. We overcome technical challenges by using *version indexes* to access tuples and *prescreening* to avoid random data access.

Operation types also affect the efficiency of query offloading. For instance, *filter* operations have proven to be the best candidates for this purpose because they are favorable to reducing data movement

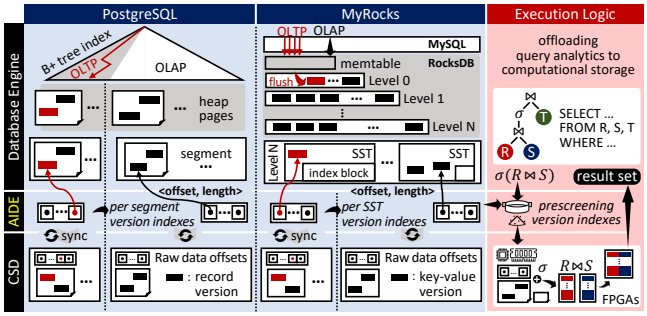


Figure 1: Overall architecture of databases with AIDE.

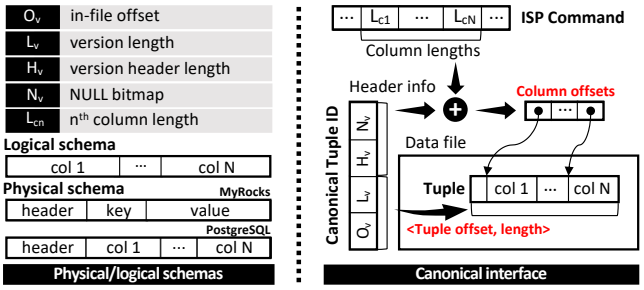


Figure 2: Canonical interface for vendor neutrality.

due to high query selectivity. Most studies primarily focused on *filter* operations and even *join* operations [10, 24, 50, 52, 55]. Such operations use simple predicates and can form vendor-neutral formats without difficulties. The *group by* aggregations can also be feasible candidates for query offloading, and in particular, hash-based aggregations can benefit from hardware parallelism [50, 52]. Although some studies attempted to offload other types of operations (e.g., sort in [55]), their operation unloading demanded substantial engineering efforts with vendor-specific storage layouts to execute the required logic. Table 1 summarizes related proposals in query offloading. We have spent time analyzing candidate relational operators by a few critical criteria: (i) popularity, (ii) resource requirements, and (iii) ease of canonicalization. We exclude some simple aggregate operations since they require vendor-specific semantics or too many resources. Therefore, our target operations are *scan*, *filter*, and *join*, all of which can readily form vendor-neutral formats. In summary, AIDE supports ISP under HTAP loads for two engines through vendor-neutral computation.

3 OVERVIEW OF DATABASES WITH AIDE

This section presents the design overview of AIDE and explains how we integrated AIDE with PostgreSQL and MyRocks having different storage formats. This overview highlights two central claims. One is vendor neutrality, where AIDE can work regardless of diverse storage layouts, and the other is a version index to deal with continuously changing data. As illustrated in Figure 1, AIDE is an intermediate layer between database engines and computational storage devices (CSDs) and is responsible for offloading all or part of query execution plans to CSDs. In particular, AIDE transforms hash joins with simple predicate conditions on alphanumeric types into a pre-defined set of primitive operations supported by CSDs and provides version indexes holding *canonical* tuple identifiers, enabling in-storage FPGAs to *instantly* spot newly modified tuple data *correctly* without understanding database-specific page layouts. In order to avoid offloading inefficient version searching to CSDs, AIDE prescreens version indexes before unloading encoded query operations to storage. We provide the design rationale for our proposed techniques.

Canonical interface for vendor neutrality. As shown in Figure 1, database engines use different underlying storage architectures, e.g., multi-segment storage in PostgreSQL and sorted string tables (SSTables) in MyRocks. They use so-called tuple identifiers and calculate column offsets using table schema encoded in a tuple header to access a tuple and required columns. So, fetching a raw tuple must pass through multiple translation steps that demand an understanding of vendor-specific layouts. For instance, PostgreSQL, a representative RDBMS, uses a combination of a page (or block) id and a slot id in a heap page as a unique tuple identifier (TID), where a slot contains a pair of an in-page offset and tuple length. In addition, accessing columns also needs the interpretation of a tuple header to correctly handle columns having NULL and variable sizes. Similarly, MyRocks manages records in the RocksDB storage engine after reorganizing MySQL record formats into RocksDB-compliant key-value formats. Such internal reformatting adds an extra translation step in MyRocks when accessing tuples/columns.

To make data access vendor-neutral inside CSDs, we propose a *canonical interface* to translate engine-specific row/column offsets into vendor-neutral formats. To this end, AIDE uses a *canonical tuple identifier* (CTID) comprising a raw tuple offset and meta-data (e.g., NULL bitmap) for retrieving column offsets from a tuple with the help of column lengths conveyed in ISP commands (see Figure 2). CTIDs are easily understandable by computational storage for spotting the latest locations for database tuples; it consists of a file offset, tuple length, header length, and NULL bitmap. With information in a CTID, CSDs calculate column offsets in a data version by summing column lengths in the given ISP command, all done without understanding vendor-specific layouts. In order to track the locations of all data versions, AIDE manages a *version index* for each file and inserts new CTIDs in version indexes when new versions arrive. Therefore, our canonical interface enables CSDs to access tuples and columns in a vendor-neutral way.

Prescreening indexes for rapid data access. Multiversion concurrency control pervading modern database systems requires that DBMSs find visible record versions before passing them to the next query execution stages. However, searching visible versions may incur a considerable delay in analytic queries when DBMSs update a large volume of data while processing OLTP workloads. Version searching overhead is a notorious issue already studied in prior work [26, 27], and unloading efficient version search algorithms still poses challenges. First, to enable computational storage to search versions, storage devices should understand vendor-specific storage layouts, breaking vendor neutrality. Even worse, if we offload such logic to devices, in-storage version searching incurs random access to data blocks, leading to worst-case scenarios. Hence, we exclude version searching logic from offloaded query execution and delegate the job to AIDE to cope with these difficulties. For this purpose, we propose a *prescreening technique* in that AIDE first filters invisible data version locators from version indexes and

rebuilds a collection of visible ones for a given query. Then, it offloads computation with the rebuilt version indexes to CSDs that will never do the version searching. The prescreening technique upholds vendor neutrality even with HTAP and accelerates query execution in CSDs by avoiding excessive access to data blocks.

4 ANALYTIC OFFLOAD ENGINE

Past research on computational storage has unveiled three design factors impacting the performance of ISP. The first factor—the selectivity of an offloaded query—decides the volume of query results, and the second factor is the access pattern inside storage that affects I/O efficiency. The last one is the programming efforts required for developing the logic in hardware accelerators. For the first criterion, ISP, even with AIDE, cannot guarantee a performance boost for all queries nor aim to achieve it mainly because modern DBMSs evolve to utilize high parallelism and asynchronous processing (e.g., prefetching). Instead, AIDE takes a best-effort strategy to avoid unfavorable cases by offloading query plans exhibiting high selectivity, e.g., a join with filter predicates. Second, in conjunction with prior work revealing that version traversal becomes a dominant cost factor in query execution, we perceive that forestalling version searching in ISP is of the utmost importance for efficiency. Third, we retain vendor neutrality when offloading computation to devices. This section discusses how we design AIDE regarding the three criteria, starting with the vendor-neutral computation.

4.1 Vendor-neutral Computation

Database systems express an analytic query as a form of relational algebra, resulting in a query plan consisting of a sequence of operations and data with proper access methods specified. As prior studies [10, 24, 54] transformed query operations into native ones supported by FPGAs, we adopt the same approach so that AIDE encodes query operations into canonical formats that the hardware accelerator can easily understand and execute efficiently. By doing this, we retain vendor neutrality for relational operations. However, preparing the target data for query operations may require understanding database storage layouts. In particular, if CSDs use the same approach as host databases, FPGAs should likewise pass through multiple translation steps to get target data offsets. We address the above issue by a canonical interface that represents engine-specific data formats as canonical forms so that computational storage accesses required tuples and columns through the lens of the canonical interface.

4.1.1 Canonical interface for tuple offsets. The canonical representation of a physical schema needs two types of offsets: tuple and column offsets. For the first type, tuple offset, AIDE extracts record/key-value offsets with their lengths and keeps these canonical tuple identifiers in a separate index, called version index. AIDE creates a version index for each data file (i.e., PostgreSQL segment and MyRocks SSTable), and the index holds in-file offsets for all data tuples stored in the file segment/SSTable. As shown in Figure 3, per file version index contains CTIDs. Managing separate version indexes has a similar design rationale to our recent work [27]. It enables computational storage to track and access the latest raw data offsets without traversing data tuples in database file segments/SSTables. Important to notice is the invariant that version

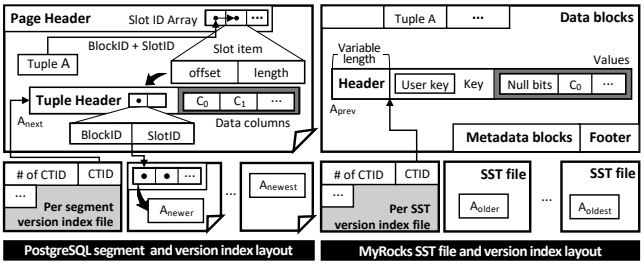


Figure 3: Storage layouts with version indexes.

indexes must hold all tuple offsets required for computation, and referred data tuples must be present in data files. Hence, AIDE must flush version indexes and their data files before query offloading for this invariant.

4.1.2 Canonical interface for column offsets. Next, AIDE translates column offsets typically embedded in a tuple header into normalized formats (i.e., an array of column sizes) and passes them to computational storage through an ISP command. Then computational storage can access target columns using a NULL bitmap in a CTID and column size information conveyed in the ISP command. Hence, the ISP command should include canonical column offsets that work correctly even with engine-specific metadata fields embedded in their physical format. As shown in Figure 4, MyRocks embeds a sequence number and a NULL bitmap between its key and the first column field. Suppose metadata fields are present in key-value data. In that case, computational storage needs to understand an engine-specific key-value layout to get a target column offset, breaking vendor neutrality of our FPGA logic.

To deal with such metadata fields, we introduce a concept of *pseudo-columns* to convert engine-specific metadata fields into one of the regular canonical columns. Then, the canonical tuple format contains more columns than the physical counterpart. To make CSDs identify the correct columns, we adjust the mapping between physical and canonical columns. Figure 4 illustrates how we remap the column order in our canonical format. When we create an ISP command for MyRocks, we should create one pseudo-column for metadata (i.e., sequence number and NULL bitmap fields) and store the pseudo-column size in the command. Later, a CSD correctly accesses the target join column by summing column lengths and header length: offset for C_{join} = H_v+$\sum_{i=0}^{join-1} L(C_i)$. So, translation from canonical into physical formats can be done inside computational storage without vendor-specific semantics.

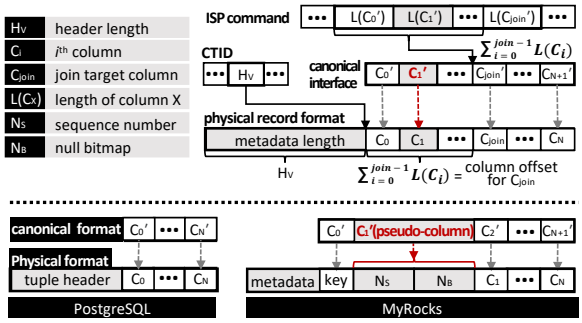


Figure 4: Translation from canonical into physical formats.

4.1.3 Version index management. Managing version indexes incurs extra contention and I/O under HTAP loads because we should hold a page latch longer while updating and syncing index entries to CSDs upon data updates. Update-heavy tables cause frequent updates on version indexes, thus inducing a longer page latch duration. To hide a long syncing delay, AIDE uses pipelined offloading by overlapping syncing activities while computational storage scans other data files. Nevertheless, we have faced unique challenges with MyRocks; i.e., there is no lifetime information in key-value data, and MyRocks runs compaction continuously under updates.

We overcome the issue by embedding lifetime information in key-value data, requiring that it be modified when a new version is committed, which determines the lifetime of the previous version. Our design induces a consistency issue between the background compaction threads and foreground update transactions if both contend on the same data file. To handle the issue, we keep track of the compaction outputs and make the updaters wait until their related compaction ends. This forceful waiting at commit can cause performance fluctuation if compaction threads are active under massive updates (see §6.4.1 for performance evaluation). It is an explicit limitation of our approach, left as future work, as it does not impair vendor neutrality.

4.2 Prescreening Version Indexes

Although we manage CTIDs in version indexes, massive updates continuously produce new ones that aggravate version searching due to increased search space. We confirm this behavior by obtaining page I/O traces of multi-table join queries under updates with and without long-running analytics on PostgreSQL. Figure 5 shows a snippet of traces for one table and implies that version searching becomes a dominant query cost since it reads data pages excessively during searching. Hence, *providing correct and necessary data locators to CSDs is at the core of the matter.* This perspective gives a few options for overcoming the technical hurdle.

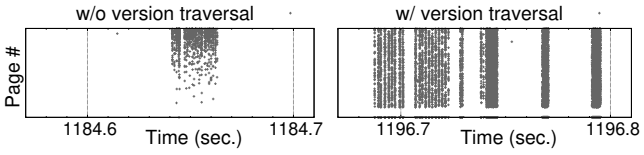


Figure 5: Page I/O traces w/ and w/o version traversal.

4.2.1 Visible data or indexes? That is the question. Despite being vendor-neutral, version lookup logic is not apposite for ISP due to excessive data access. More crucial to excluding the logic from the main body of offloaded analytics is how to provide visible data. Fortunately, version indexes already hold raw data offsets with visibility information, enabling AIDE to prescreen the indexes to filter invisible data offsets. This prescreening procedure can pass all valid data offsets to storage before offloading the main computation body. The prescreening of input data offsets in version indexes is conceptually the same logic executed by traditional MVCC databases. It inspects CTIDs in version indexes and filters out tuples invisible to analytics by query snapshots. Validating CTIDs requires two timestamps — *min* and *max* — for each tuple, and AIDE embeds meta-data in in-memory version index entries.

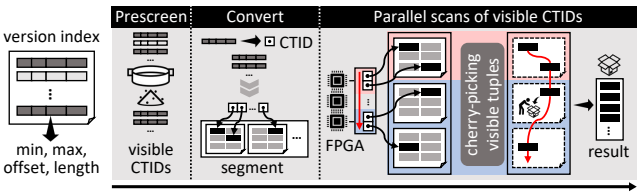


Figure 6: Prescreening enables parallel file scans in CSDs.

The host-side prescreening opens opportunities for accelerating ISP. First and foremost, file-wise screening enables CSDs to use sequential access to the target data file. Since the FPGA platform loads an entire segment/SSTable into its dedicated memory and performs operations in memory, it *forestalls random access across multiple files.* File-wise prescreening further allows FPGA to scan data files in parallel, enabling *parallel file scans* (see Figure 6). Second, version inspection is limited to each data file to screen visible CTIDs from the version index. So, a prescreened version index can confine in-storage scanning to its segment/SSTable, thus avoiding random data access across multiple data files.

4.2.2 Prescreen-offload-execute model. Combined with vendor-neutral computation, the prescreening technique can form an efficient offloading framework. We explain the overall sequence of query offloading using a simple join with filter predicates. The distinction between legacy engines and the engines with AIDE arises after the legacy query optimizer produces a query plan. With a given query plan, the engines with AIDE adopt the so-called *prescreen-offload-execute* model. In the prescreening stage, we must prescreen version indexes if ever modified since the last prescreening. To further optimize prescreening, AIDE internally manages a file-wise minimum transaction ID for each data file and a minimum value for a group of in-file data blocks. If a table has many data versions, AIDE can efficiently decide whether or not to prescreen an entire data file or a portion of page groups. This pruning strategy is effective for tables undergoing massive updates (see Sections 6.3 and 6.4 for more details).

In the last stage, a CSD executes query operations by first performing parallel scans of target files using the given version indexes. It applies filter predicates to sift valid tuples satisfying the given predicates. It then builds a hash table using filtered tuples and finds matching tuples in the probing phase. If spilled, FPGA saves intermediate or final results to a hidden result file during the probing phase. Unlike the vanilla engine, AIDE always enables sequential data access inside computational storage, thus escaping performance disruption even with massive updates on target tables. In HTAP environments like TPC-CH [12], we offload a subtree of an original query plan tree that would suffer from redundant, random access under massive updates but exhibit high selectivity; for example, join queries involving update-intensive dimension tables (i.e., `stock`) and tables with filter predicates.

4.3 Canonical Interface for Query Offloading

Query offloading in AIDE requires two types of information: encoded query operations and canonical tuple/column offsets. Our prototype system uses OpenCL commands to communicate with computational storage to either pass such information to storage or receive query results. AIDE plays a translator for its host database

engine to unload computation with canonical tuple/column formats to its underlying computational storage. The information delivered to storage consists of descriptions for operations and file descriptors for data files and their associative version indexes. Overall, AIDE should pack all these parameters into a contiguous memory and submit OpenCL commands to CSDs; then, the FPGA receives the packed message to interpret for initiating offloaded query operations. For obtaining query results, computational storage saves the final results (i.e., tuples) into separate temporary files for given tables. The layout of the result file is as follows; the first 6-byte fields contain the total number of tuples and the maximum tuple length, and the rest of the file space stores result tuples.

5 IN-STORAGE QUERY PROCESSING

This section presents our implementations of required functions for in-device analytics in commodity CSD that supports the latest Xilinx Vitis™ development platform, allowing users to write functions executed on FPGA using its High-Level Synthesis (HLS) tools. We provide the key details for understanding our implementations of vendor-neutral logic and its pipelined workflow to accelerate query analytics. In essence, the Vitis OpenCL host code linked to AIDE controls our FPGA logic in this CSD. Note that AIDE only supports exclusive query offloading, where our CSD executes one query plan at a time without accepting multiple query plans simultaneously.

5.1 Hardware Accelerator Logic

5.1.1 Core components of computational storage devices. Figure 7 depicts our proof-of-concept CSD and its in-device analytic components tested in this work. Our CSD comprises an NVMe SSD and an acceleration card, including FPGA logic and memory (4 GiB). A tailored internal PCIe switch connects PCIe endpoint devices and dispatches host calls to the correct target (omitting the PCIe switch for simplicity). In particular, it delivers I/O requests sent from the host NVMe driver to an SSD; likewise, it passes acceleration calls dispatched from Xilinx Runtime (hereafter noted as XRT) to the embedded FPGA card. XRT is implemented as a device driver and provides the OpenCL-compliant software interface facilitating communication between the host application and the acceleration kernels (functions compiled for FPGAs) deployed on the FPGA card (i.e., Xilinx Kintex™ Ultrascale+ FPGA). Executable device binary downloaded to FPGA logic includes acceleration kernels responsible for native operations, while OpenCL commands take control of memory management and data movement between the host and hardware accelerators. Essential among many functionalities of XRT is the task of orchestrating the execution of triggering, sequencing, and synchronizing kernel computations.

5.1.2 Control and data plane. According to the OpenCL programming model, we split ISP logic into host code and device code. The host code, written in OpenCL, is linked to AIDE. At the same time, the device code implementing filtering and hash join, written in C, is compiled by Vitis HLS tools into an executable device binary that runs within the programmable logic of FPGA. Communication between these two, including control and data transfer, occurs across the PCIe bus for the XRT platform. Also, global memory in the acceleration card is shared and used to transfer data between them. Note that the host code takes control of issuing I/O commands and

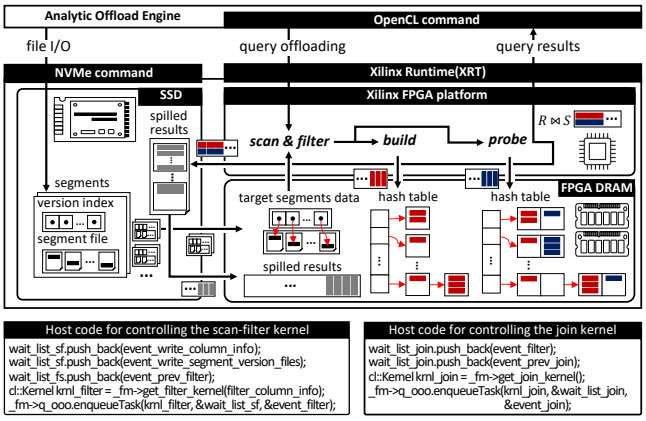


Figure 7: In-storage query processing logic.

loading data blocks into FPGA DRAM. Meanwhile, device code (i.e., FPGA logic) operates data in its memory buffer and does not have to understand the host file system or storage blocks.

As depicted in Figure 7, AIDE invokes OpenCL commands to guide XRT to transfer the input parameters from the host database to FPGA memory or move input data from SSD to FPGA memory. Such commands also trigger sequenced invocation of the kernels in the executable device binary that computes and stores results back to the FPGA memory. The capacity limit of FPGA memory causes kernels to spill intermediate or final output into a hidden file in storage. It later would be loaded to FPGA memory upon AIDE's request for query outcome, copied into the host memory, and consumed by host databases, i.e., PostgreSQL and MyRocks.

5.1.3 Hardware acceleration in action. The OpenCL host code attached to AIDE controls the overall sequence of query offloading. Upon a query request, AIDE creates memory buffers on the device and copies input parameters (i.e., a list of join clauses, filter predicates, and file descriptors of data files with their associative version indexes) into them. It also loads data files and their associative version index files into global memory. Then, from the executable device binary, it creates program and kernel objects on FPGA, sets the kernel arguments, and then triggers their sequenced execution. While reading data from global memory, scan-filter and join kernels perform native operations on the data: executing scan-and-filter, building a hash table using filtered tuples, and probing the hash table to find matching tuples. Join results are stored back to global memory or spilled into an SSD under memory pressure. Such sequenced execution continues as long as data and version index files arrive. Later, AIDE requests query results residing in global memory or SSDs.

5.2 Pipelined Query Processing in FPGA

The core of vendor-neutral computation in our CSD is the pipelined architecture of *scan-filter-join* operations that we implemented as separate FPGA kernels. The pipelined architecture illustrated in Figure 7 does not have a one-to-one correspondence with core FPGA kernels since it includes the stage of loading target files into FPGA global memory, and more details explaining the mismatch will follow after covering core kernels. AIDE packs the kernel input parameters in a message as an ISP command and loads it into global

memory before initiating kernels. We explain each kernel operation first and describe the pipelined processing architecture next.

5.2.1 Scan-filter kernel in action. The first kernel performs dual roles—scan and filter operations—and we embody these into a single scan-filter kernel. The scan logic operates on data files with their version indexes loaded from the SSD into global memory; it can find the correct location for each visible tuple within the file boundary. The scan operation feeds visible tuples to the next *filter* operation that applies user-defined predicates to the fetched tuple stream. For all valid tuples, the filter operation adopts the late materialization technique; it stores the location, instead of an entire tuple, and the column value required for the next *join* operation so as to save global memory and reduce the number of memory accesses. So, the scan-filter kernel passes tuple meta-data to the final join kernel.

5.2.2 Join kernel in action. The following join kernel performs the so-called *build-probe* phases that build a hash table using the left (i.e., build) table and probe to find matching tuples. After the scan-filter kernel is complete on time for the join stage, the join kernel starts probing the constructed hash table using newly arriving join columns and then builds a new hash table using matching tuple data in this step, except the last join step. Note that the join kernel supports multi-table join and assumes a hash table built in the previous step. In the case of the first join step with no previous hash table, it considers all tuples for probing to find a match. We set a pre-defined threshold for each data structure in memory for fair usage of limited global memory. If the memory usage of the hash table exceeds its limit (128 MiB in our prototype), query offloading will fail with a join memory exhaustion error. For the same reason, when join results require more memory beyond the buffer limit (480 MiB), the join kernel stores spilled results in a hidden file.

5.2.3 Put all core kernels into pipelined processing. AIDE exploits pipeline parallelism since the sequential execution naturally leads some ready kernels to wait around for their input. Our PoC CSD uses a three-stage pipeline consisting of load, scan-filter, and join. The first load stage is a simple data movement from SSD to FPGA memory that requires no separate kernel. Hence, only the last two stages have matching kernels. We synthesize those kernels into a single executable binary, thus making them co-reside on the same FPGA. To allow simultaneous execution, we examined data dependency across stages. The last two stages must wait for their preceding stage to complete, while the first load stage can run independently. We show the host code snippets that orchestrate the pipelined execution of FPGA kernels on the bottom of Figure 7, omitting device code for space limitation. Note that we created the command queue (i.e., `_fm->q_ooo`) with out-of-order execution (i.e., `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`) enabled to allow concurrent execution of kernels having no dependency.

5.2.4 The pros and cons of our approach. There are some pros and cons of our approach. In the case of best-fit queries (refer to §6.2 for more details), '`build`'-side data files would be relatively small compared to '`probe`'-side data files in the hash join stage, and filter predicates would filter out a majority of tuples. So, all three stages would take considerably less time for the build phase than the probe phase. Moreover, FPGA memory access is far more time-consuming than computation. The scan-filter stage incurs more random memory access and takes longer than the compute-intensive join stage. Thus, parallelism is logically proportional to the number of data files if we start the scan-filter stage at each step.

However, our PoC CSD imposes strict limitations on the global memory and programmable logic blocks, 4 GiB and 300K LUTs respectively. Considering the required memory for each data file (i.e., up to 1 GiB) and in-memory data structures, such as hash table and join result buffer, it allows up to two data files for concurrent processing. Moreover, the number of logic blocks required for multiple kernel instances exceeds the PoC device's capacity. Such restriction of our mid-range FPGA device limits the concurrency level to two, with a single instance of each kernel executing. We create a single out-of-order command queue for the parallel execution of these kernel instances where enqueued kernels can go out of order. To this end, we use the OpenCL event (cl::event) to make them wait for their preceding stage(s) to finish.

Despite all our effort in developing FPGA kernels, FPGA-based ISP has a few critical limitations. Since modern databases have adopted asynchronous I/O and parallel processing (e.g., parallel scan) to reduce (or hide) the speed gap in the memory hierarchy, many ISP proposals, including AIDE, with general-purpose designs may still possess I/O or memory access delays. Due to these limitations, the AIDE-enabled ISP may not guarantee better processing speed than the vanilla system when operating conditions do not meet our criterion (i.e., version search overhead). However, it will show a breakeven point as random, redundant access starts dominating the cost for query processing (see Sections 6.3 and 6.4 for more details). We believe this can be a dilemma between DBMS-dependent or DBMS-oblivious.

6 EXPERIMENTAL EVALUATION

6.1 Evaluation Setup

As a prototyped system, we implemented AIDE for PostgreSQL-12.0 and MyRocks of Facebook MySQL 5.6. Also, we wrote FPGA kernels, and use the same kernels for both database engines. Our server features two AMD EPYC 7003 processors, each having 64 cores and 256 MiB cache. It has 128 GiB memory. We also use Samsung SC1733 CSD having an FPGA with 300 MHz clock frequency and 300K LUTs. Since our CSD has limited FPGA memory (i.e., 4 GiB) compared to sufficient memory in our server, we intentionally restrict available DRAM and database buffer sizes to emulate larger-than-memory analytics environments. To limit host memory, we use the `cgroup` feature of the Linux kernel.

Workloads. We use two benchmark tools to measure various performance metrics on our server: sysbench-1.0.20 benchmark [1] with our synthetic multi-table join queries and TPC-CH [12], a mixed workload containing both TPC-C and TPC-H. For TPC-CH, we adopted the open-source tools [11] released by Citus Data [13] for running CH-benCHmark with HammerDB [2]. This is the most common benchmark to evaluate HTAP systems, and uses a hybrid schema of all TPC-C tables and some TPC-H tables. It runs all TPC-C transactions without changes and TPC-H queries with modifications. We only create a small dataset with scale factor 2 (SF=2), equivalent to 1.6 GiB, since the dataset will grow quickly as update transactions run. In addition, we introduce special update transactions to create situations that produce a continuous stream

of data versions from the stock table, which increments and decrements stock level count without infringing TPC-CH specifications. To this end, a group of dedicated threads only run that transaction continuously, and we call this workload TPC-CH+. Since their update throughput is around 34kTps, we expect that initially, AIDE may not be helpful to ISP for both engines. However, it will later show the breakeven point where ISP can be faster than vanilla systems that spend time in version searching.

For the sysbench experiments on each engine, we create 12 tables, each with 100k tuples, and use OLTP transactions updating tables. We then open a query that internally performs multi-table joins (randomly selected four tables) continuously. As time elapses, join queries should act on multiple data versions in the vanilla system, while AIDE-enabled ISP only accesses required visible versions during join processing. The setting is similar to the one used in the prior work [27], so we reconfirm the same phenomenon and see whether our approach can address it.

Common database configurations. We have set common configurations to allow more updates on tables. In PostgreSQL, we use large log files to suppress checkpoint activities and relax strict durability by adopting "*asynchronous commit* [44]". Also, to reduce unwanted contention between worker processes, we disable the PostgreSQL parallel query [43]. All analytic queries in TPC-CH+ run under the isolation level `REPEATABLE READ`. For MyRocks, we disable binary logging used for replication. We also relax strict durability by avoiding log flushing at transaction commit.

AIDE-specific database configurations. The cost-based optimizers of both engines try to choose the best plan, but at present may not be perfect for AIDE. When generating query plans, they are unaware of properties related to ISP (e.g., operational costs for version search and underlying CSD) and neglect them. To enable our AIDE-based ISP, in PostgreSQL, we enforce the hash join algorithm with the sequential scan access method. We deploy an extension module (i.e., `pg_hint_plan` [3]) and generate apposite query plans to ISP. By adding hinting phrases to the target queries, we can reorder relational operations and change access methods for tables. For example, we attach the following hint prefix to the Q8 query of TPC-CH+ to enforce the hash join algorithm with the specified join order: `/*+ Leading((stock item)) HashJoin(stock item) */`. Of course, the vanilla PostgreSQL optimizer generates a different query plan. For MyRocks, at the start of each query session, we set a session variable for the target table to offload. AIDE-enabled MyRocks generates a query plan where the table retrieved from the session variable would be the first of the join order. Also, to enforce predicate filtering and join by ISP, it transforms index seeks for target tables into table scans.

6.2 Workload Characterization

The characteristics of target query plans offloaded to our CSD substantially impact the performance of ISP. Although TPC-CH+ faithfully emulates HTAP workloads, not all the queries deserve ISP because queries may include static (or insert-intensive) tables. Queries also specify various join types (e.g., full/anti/semi-join) or memory-intensive operations (e.g., sort, GROUP BY, or ORDER BY) that we did not embody in our kernels. To get a better understanding of ISP with various query plans, we add hints to all queries in TPC-CH+ and categorize them into four groups: G1 (most favorable), G2, G3, and G4 (ill-suited to ISP) based on the 'EXPLAIN' outputs from vanilla PostgreSQL. Then, we offload sub-query plans from the first three groups and see how good or bad the efficiency of query offloading is. None of the prior studies have analyzed the in-situ processing of query analytics in HTAP environments.

6.2.1 `Join` *by hash-based equijoin.* To maximize the performance improvements while attaining better performance isolation by unburdening host database systems, we use three empirical criteria when identifying good candidates: i.e., type of join operations, frequency of updates on target tables, and the presence of filter predicates. The criteria guide our prototype to support the equijoin condition wherein FPGA scans inner and outer tables sequentially. However, as opposed to prior work, we do not aim to offload a single table scan since simple in-situ table scans would not give significant benefits. Unlike table scans, joins are resource-hungry operations; the reduction in a result set during in-storage join processing can propagate up to the root of query plan trees, leading to reduced query results. Also, database systems have used software optimization techniques, such as pushing down filter predicates, in the early stages of query processing. We, therefore, prefer to choose sub-plans having equijoin operations.

6.2.2 `Join` *with update-heavy tables using filter predicates.* It is well-known that query selectivity profoundly impacts query processing. Specifically, the presence of massively updated tables and filter predicates substantially affects the efficiency of in-storage table scans. When join tables have neither static (or insert-intensive) properties nor filters to be applied, one cannot expect a considerable reduction in the result set and cannot achieve much gain accordingly. Considering the limited computing power and memory in FPGA cards and the considerable cost of deploying them (e.g., initialization, communication, and data transfer), the gains will unlikely outweigh the price we pay. However, suppose we enhance a reduction factor by eliminating the hefty version search logic and additional filter predicates occurring during in-storage join processing. In that case, AIDE can produce the desired result set that would decrease, owing to ISP. In TPC-CH+, queries having join conditions involving fact and dimension tables with massive updates and filter predicates are suitable candidates for query offloading.

6.2.3 Extract target `join` *sub-trees.* We develop different strategies for identifying a candidate sub-tree to offload, since unfortunately the only join method in MySQL 5.6 version is nested-loop method. In PostgreSQL, AIDE starts a plan tree traversal in reverse order. When meeting any join node whose type is not inner hash join, it stops and assesses how suitable this sub-tree is for offloading by considering the criteria described above. We focus on the join node at the tree's top-level obtained from the reverse order traversal. If any of the join tables is update-heavy and has filter predicates to be applied to, this query must be apposite for offloading. By contrast, for MyRocks, we guide AIDE on how to generate a query plan using MySQL system variables. By referring those variables, it sets the first of join order and adjusts the number of tables to offload. Also, it converts index seeks for target tables into table scans.

Table 2: Categories (un)suitable for query offloading

Group	Queries	Offloaded sub-plans	FR†
G1	Q2, 5	$\sigma_{\text{r\_name like 'Europ\%'}}T_r \bowtie T_n \bowtie T_{sup} \bowtie T_{st}$	0.80
	Q8	$\sigma_{\text{i\_data like '\%b'}}(T_i) \bowtie T_{st}$	0.98
	Q9	$\sigma_{\text{i\_data like '\%BB'}}(T_i) \bowtie T_{st}$	0.99
	Q11, 21	$\sigma_{\text{n\_name = 'Germany'}}(T_n) \bowtie T_{sup} \bowtie T_{st}$	0.98
	Q20	$\sigma_{\text{i\_data like 'co\%'}}(T_i) \bowtie T_{st}$	0.99
G2	Q7	$T_{sup} \bowtie T_{st}$	
	Q15	$T_{st} \bowtie T_{ol}$	
	Q16	$T_i \bowtie T_{st}$	
G3	Q3, 10, 12, 14, 17, 18, 19		
G4	Q1, 4, 6, 13, 22 (Not evaluated)		

※ T_α is a table in TPC-CH+ benchmark, where $\alpha \in$ {'r' (region), 'n' (nation), 'sup' (supplier), 'st' (stock), 'i' (item), 'ol' (order_line)}.

† FR: Filtering ratio of a predicate

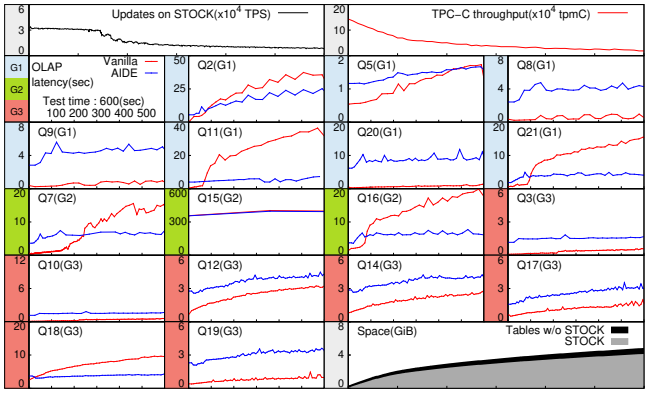
How well query characteristics conform to our empirical criteria determines the speed-up by our ISP. For evaluation, we apply our criteria to the whole 22 TPC-H queries from CH-benCHmark and categorize them into four groups. As shown in Table 2, only queries that belong to the first two groups, i.e., G1 and G2, become good candidates for offloading. After classifying them, we obtain seven and three queries for each group, respectively. The queries in G1 would be the best fit for ISP because they include a representative `stock` table under massive updates as well as a small dimension table with a highly-selective filter predicate (e.g., the `item` table), filtering out at least 80% of tuples. Such queries pass reduced join results to its parent node for further processing. In contrast, AIDE would avoid or ignore queries from the last two groups because they seldom satisfy the first or the second criterion.

6.3 TPC-CH+ Workloads

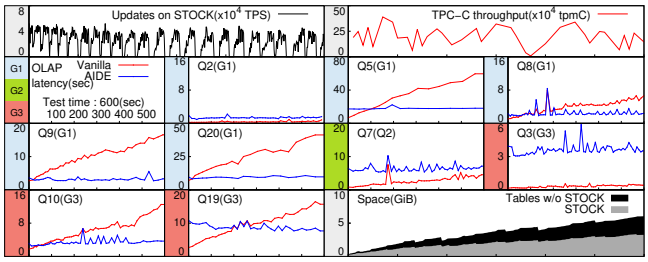
In this section, we run TPC-CH+ workloads and use the HammerDB-4.4 benchmark. Note that TPC-H queries in G4 have either futile simple scans or join types unsupported by AIDE and have not been evaluated. We use a small dataset (SF=2) to see the behavior and configure our server to limit physical memory to 8 GiB. We set a 2 GiB buffer pool for PostgreSQL and MyRocks. In this experiment, we use dedicated workers to run TPC-C and TPC-H; 12 threads for TPC-C, 44 threads for TPC-H queries—half of them run within a long-lived transaction to intentionally emulate larger-than-memory analytics. In addition, we assign eight threads for updating the `stock` table, which impacts overall performance. As time elapses, two noticeable changes occur: (1) insert-intensive tables (e.g., `order_line`) grow due to newly inserted tuples, and (2) the size of the `stock` table also increases because of continuously generated data versions.

Our TPC-CH+ workloads overburden both engines and incur harmful performance interference. For evaluation, we adapted the TPC-CH+ schema to make as many queries runnable in our vendor-neutral CSD as possible, e.g., transforming string matching on a fixed length prefix into an equality condition on a newly added column storing a character value. This temporary expedient may harm MyRocks since it shows immense overhead in join processing on a non-indexed column.

6.3.1 Performance metrics of vanilla PostgreSQL/MyRocks. We first run TPC-CH+ on vanilla engines with a small dataset (SF=2). We measure three performance metrics: the number of data versions generated, latency for each TPC-H query, and space overhead. Although we intentionally limit physical memory to emulate larger-than-memory analytics, three metrics look normal until the version



(a) Performance of PostgreSQL under TPC-CH+

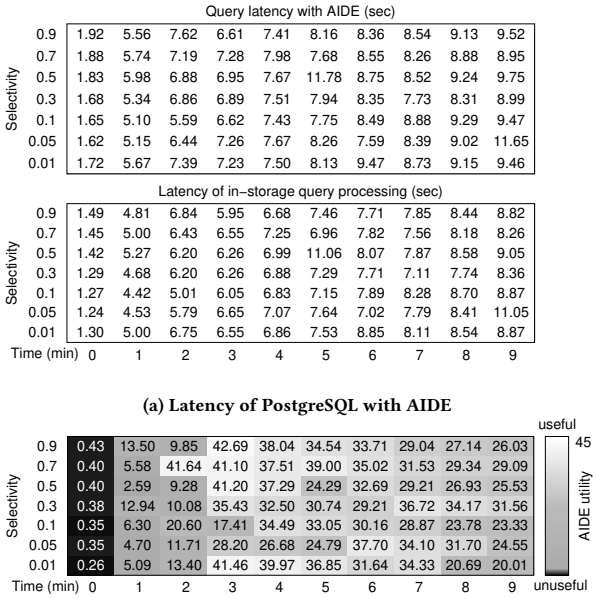


(b) Performance of MyRocks under TPC-CH+

Figure 8: Performance of the engines under TPC-CH+.

search on massive data versions overburdens the database buffer. As shown in Figure 8a, seven queries in G1 and G2 (i.e., Q2, Q5, Q7, Q11, Q15, Q16, and Q21) show increasing latency in vanilla PostgreSQL as update transactions produce data versions in the `stock` table. The remaining queries in G1—Q8, Q9, and Q20—are noticeable since they exhibit slowly increasing query latency. For these queries, the vanilla PostgreSQL optimizer generates plans performing join operations against large tables first, which passes a reduced result set to the upper-level join with the `stock` table. In practice, the highly-selective inner join against the largest `order_line` table returns a tiny result set (100 ~ 200 records). In this case, the version search overhead for matching tuples from the `stock` table would not be noticeable and took less than 2 *sec* in our evaluation (SF=2). Other queries in G3 are not acting on the `stock` table, but are only affected by insert-only tables (e.g., `order_line`). Among them, the latency of Q18 is noticeable. In this case, the PostgreSQL optimizer's large-tables-first strategy backfires, and non-selective hash joins with sequential scan methods to the largest (and increasing) `order_line` and `orders` tables are performed first.

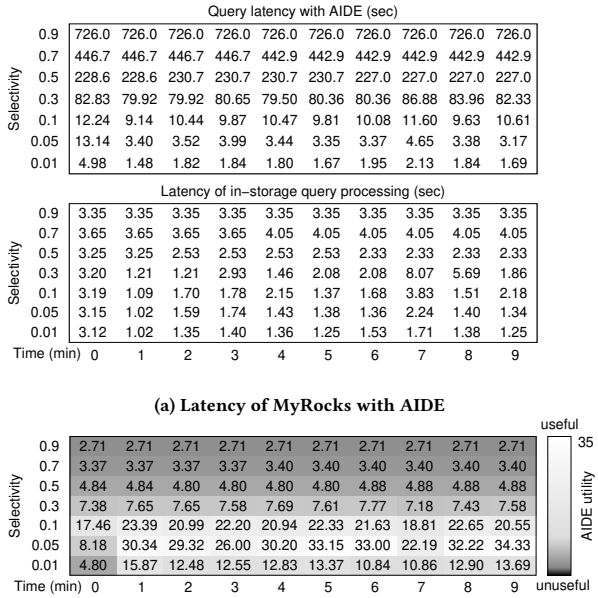
Vanilla MyRocks shows similar behaviors, except for noticeable differences between the two engines (see Figure 8b). Since MyRocks internally relies on the RocksDB storage engine, it has to reformat key-value data into MySQL record formats. When MyRocks performs nested loop join algorithms, it has to fetch and reformat key-value data streams more than necessary, even with queries having high selectivity. Such queries are Q10 and Q19 in G3, which show increasing latency in vanilla MyRocks. Another exception is Q2 which runs much faster compared to vanilla PostgreSQL. Other queries in G1 and G2 show similar behavior as we have seen in vanilla PostgreSQL. As mentioned earlier, some queries — Q12, Q14,



(a) Latency of PostgreSQL with AIDE

(b) Normalized latency of vanilla PostgreSQL ($\frac{Latency_{vanilla}}{Latency_{AIDE}}$)

Figure 9: Performance of multi-table joins on PostgreSQL.



(a) Latency of MyRocks with AIDE

(b) Normalized latency of vanilla MyRocks ($\frac{Latency_{vanilla}}{Latency_{AIDE}}$)

Figure 10: Performance of multi-table joins on MyRocks.

and Q17 — experience colossal delays due to MyRocks' join processing overhead on our non-indexed columns, so we omit comparison results with vanilla MyRocks here. Note that the vanilla engine's metrics — update rate on the `stock` table and tpmc — are similar to the AIDE-based engine, so we only show numbers with AIDE.

6.3.2 Performance metrics of the engines with AIDE. Next, we run the same test on the engines with AIDE. Since AIDE accelerates the execution of sub-plans acting on update-intensive tables by offloading join operations to storage, we show the query latency for representative queries primarily concerned when the dataset has many data versions in the `stock` table (i.e., at time 600 *sec*. As shown in Figure 8a, queries in G1—Q2, Q5, Q11, and Q21—and queries in G2—Q7, Q15, and Q16—outperform vanilla PostgreSQL by up to 6.76× (Q11). Except for Q2 and Q5, AIDE can sustain their processing latency, although we enforce a less effective query plan to PostgreSQL. Noticeable are their latencies with non-optimal query plans that we force PostgreSQL to use by joining the `stock` table first. The latency tells that the benefit of running a non-optimal query plan to avoid version searching overhead sometimes outclasses an optimal plan generated by the standard optimizer, unaware of increased version search overhead. The results of MyRocks with AIDE are similar to PostgreSQL. AIDE-enabled MyRocks can sustain query latency better than AIDE-enabled PostgreSQL, regardless of versions and selectivity. What is interesting to notice from our experiments is that, unlike PostgreSQL, selectivity substantially affects vanilla MyRocks due to its inherent internal reformatting overhead.

6.4 In-depth Analysis with Multi-table Joins

6.4.1 Experimental setup. In this section, we perform an in-depth analysis by running synthetic multi-table join queries with sysbench OLTP workloads. We created 12 tables, populated each table with 100k tuples, and set 2 GiB for a database buffer pool with 8 GiB of physical memory restricted by `cgroup`. We use 48 dedicated worker threads to update target tables, committing around 200kTps. For running synthetic OLAP queries, we use 20 OLAP clients, again half of which run multi-table join queries on randomly selected four tables within an open transaction to emulate long-running analytic queries, while the other half run each query within a single transaction. We adjust a predicate condition imposed on a table to measure the performance with varying selectivity. The following is an example of our synthetic multi-table join queries:

```
SELECT SUM(sbtest1.k + ... + sbtest4.k)
FROM   sbtest1, sbtest2, sbtest3, sbtest4
WHERE  sbtest1.id = sbtest2.id AND sbtest2.id = sbtest3.id AND
       sbtest3.id = sbtest4.id AND sbtest1.k <= 10 /* PostgreSQL */
WHERE  sbtest1.k = sbtest2.k AND sbtest2.k = sbtest3.k AND
       sbtest3.k = sbtest4.k AND sbtest1.k <= 10   /* MyRocks */
```

We only enable the hash join algorithm to guide the PostgreSQL optimizer to generate what ISP-enabled AIDE expects. The experiment runs for 10 minutes by running all transactions/queries altogether. For MyRocks, we use different configurations. MyRocks supports a block nested loop join algorithm that takes significant time in reformatting key-value data into MySQL record formats; in particular, when we perform multi-table joins on non-indexed columns, the overhead becomes immense. So, we decide to use non-indexed columns (i.e., *k*) as join targets. In this case, AIDE can help substantially reduce the result tuple set digested by MyRocks.

6.4.2 Performance metrics of PostgreSQL with AIDE. As we have seen in the previous results with TPC-CH+ workloads, AIDE-enabled in-storage query processing initially has longer latency than the vanilla engine. This result explains why we could expect little gain from offloading joins whose input tables are barely different in size. In this case, the gains will unlikely outweigh the cost we pay, considering various FPGA kernel overhead (e.g., initialization and

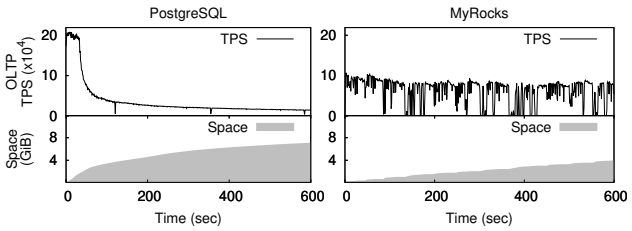


Figure 11: Update throughput and space overhead.

communication). However, Figure 9a shows that AIDE-enabled ISP can sustain the query processing delay, although update transactions produce a large volume of data versions piled up in heap pages; hence, it isolates multi-table joins from transactions/analytics running in the host database.

As queries get more data from ISP (i.e., higher selectivity value), AIDE takes longer to complete the post-processing of result tuples, but in-storage processing time would stay the same. As versions pile up, the effect of AIDE's prescreening proves its worth in eliminating redundant, random access to data pages, thus offsetting the fixed cost for ISP. To verify our claim, we also measure the latency of in-storage processing and plot results in Figure 9a. The general tendency is that computational storage sustains the time to complete offloaded query operations throughout all experiments, except that we can see a non-trivial increase in latency as time elapses due perhaps to internal throttling mechanisms (see §6.4.6).

6.4.3 Performance metrics of MyRocks with AIDE. As shown in Figure 10, the normalized latency shows a steep fall upon selectivity increase. This behavior is due to the overhead of MyRocks join processing explained in §6.4.1. Nevertheless, AIDE performs better in all spectrums of selectivity since AIDE-based ISP reduces the intermediate join set size. In-storage processing time shows a similar tendency as in Figure 9a, although our CSD acts on SSTable files whose underlying layouts differ from PostgreSQL segment files. The latency of the vanilla engine is around 23 *sec* with a selectivity value of 0.01. However, the latency grows steeply to 103 *sec* with selectivity 0.05 and up to 1966 *sec* with selectivity 0.9. As selectivity increases, the increased join set size attenuates the reduction effect of AIDE-enabled ISP. Thus, the query latency of AIDE increases, and accordingly, the gap between the vanilla and AIDE narrows.

6.4.4 The cost of in-storage processing. As shown in the left side of Figure 11, OLTP transactions initially sustain a decent update rate when they start running with analytic queries. As our synthetic multi-table join queries start, growing version space in heap pages impacts crucial performance metrics. First, short-running queries or transactions slow down due to increasing version traversal time in PostgreSQL; short transactions spend excessive time traversing from the oldest version to the recent tuple. Short queries have longer latency for the same reason. Second, PostgreSQL suffers from space overhead due to delayed vacuuming. In all environments, our synthetic analytic queries have growing latency values as data versions pile up. Update throughput also degrades since short transactions spend more time traversing version chains.

The right side of Figure 11 clearly shows the overhead for maintaining the version index in MyRocks. In MyRocks, copies of a data version may spread across multiple SSTables due to continuous

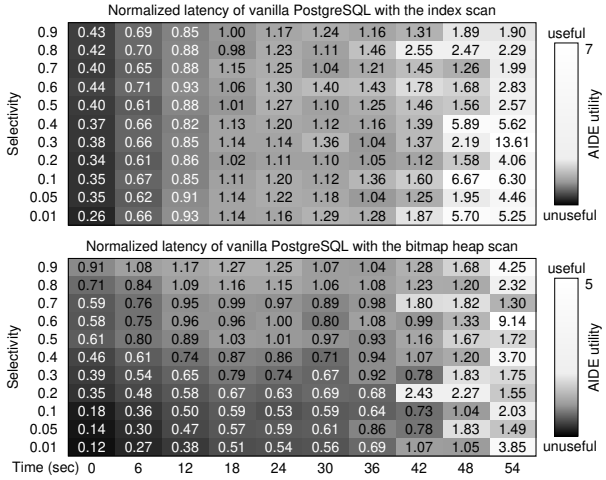
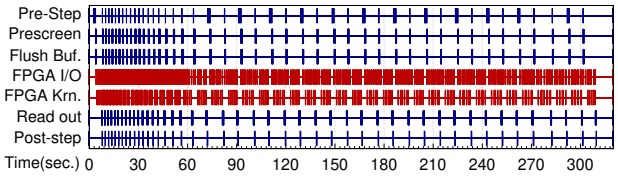


Figure 12: The utility of AIDE showing breakeven points.

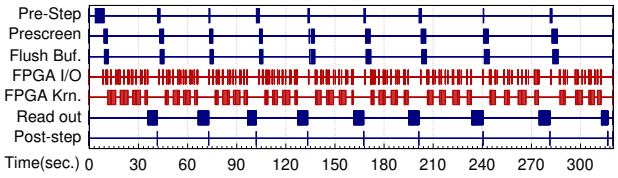
compaction. Since we must modify the version index on updates, we must mutate data copies across the RocksDB storage engine to preserve consistency. This design breaks the invariant that SSTables are immutable, inducing a concurrency issue between the background compaction threads and the version index writers. To handle the issue, we keep track of the compaction outputs and make the writers wait until their related compaction ends. This forceful waiting causes the sharp fluctuation in Figure 11. This outcome is an explicit limitation of our approach, and we leave it as future work as it does not impair vendor neutrality.

6.4.5 AIDE vs. other access methods. We use vanilla PostgreSQL with the same workload and configuration to quantify the utility of the AIDE-based ISP framework. For this experiment, we configure the vanilla engine to use two well-known table access methods—`index scan` and `bitmap heap scan`—if possible. As shown in Figure 9b, the vanilla engine's overall query latency behavior worsens as expected; in particular, PostgreSQL with AIDE is better than the vanilla engine (i.e., normalized latency is greater than 1) after 1 *min*. To get fine-grained results on a timeline showing breakeven points before 1 *min*, we show normalized latency results of vanilla PostgreSQL with the two access methods in Figure 12. The results clearly show the pros and cons of the two access methods; the `bitmap heap scan` shows strengths with low selectivity values, whereas the `index scan` works well with high selectivity values. Despite their merits, PostgreSQL with AIDE outshines all of them after breakeven points, where the latency for vanilla PostgreSQL steeply increases due to the dominant version search cost. The results imply that query optimizers can hit sweet spots to get the best of both approaches by dynamically offloading sub-plans to computational storage, provided that crucial information is available.

6.4.6 Where does the time go? We use Xilinx performance profiling toolkits (i.e., Vitis Analyzer [4]) to peek inside the FPGA to get more accurate timing information and runtime profile. The best scenario is that all are busy doing productive activities, but reality depends on workloads. Figure 13 breaks down ISP when a single sysbench query has been conducted twice on different datasets of 100k and 1M tuples, respectively. In the first `Pre-step` stage, AIDE



(a) Multi-table joins on tables with 100k records per table (selectivity: 0.1)



(b) Multi-table joins on tables with 1M records per table (selectivity: 0.1)

Figure 13: The breakdown of ISP on PostgreSQL.

loads parameters into FPGA global memory and then feeds kernels with target data and associative version index file descriptors after forcing them to storage (`Prescreen` and `Flush Buf.`). After reading each pair of data and version index files from SSDs (`FPGA I/O`), AIDE enqueues kernels, which are executed through a command queue (`FPGA Krn.`). When PostgreSQL later starts a sequential scan on the offloaded join tables, AIDE loads ISP results into host memory for further processing (`Read`), and PostgreSQL executes the remaining steps in the original plan (`Post-step`).

This experiment is to confirm that the dataset size affects the performance of ISP proportionally. As we use large datasets, ISP needs more time for I/O on both host and FPGA sides. Long execution times may lead to power competition between SSDs and FPGA. Our PoC device internally has a power capping mechanism; thus, both hardware components start decreasing I/O performance and FPGA frequency to stay within the given threshold. Figure 13a shows that the time spent on FPGA activities increases as time elapses (from 1.3 *sec* to 8 *sec*), whereas other AIDE activities spend almost the same time. When we increase the dataset to 1M tuples, Figure 13b shows that FPGA activities already spend 23 *sec* when the query starts, which is twice as much as we expect. The time duration increases to 34 *sec* at the end of the experiment. With limited profile information, we only conjecture that *internal throttling mechanisms* restrict FPGA activities unfavorably. Device vendors' in-depth studies are more than welcome.

6.4.7 AIDE vs. high-end vanilla PostgreSQL. To get more insight into the value of the AIDE-based ISP framework, we measure the latency of vanilla PostgreSQL on a high-end server, using 100 GiB of shared memory without resource restrictions. We also turn on the parallel join to fully utilize all workers, and there is only one background long-running analytic query. In this setting, vanilla PostgreSQL uses 24 parallel workers to run a single join query with all heap pages in memory. Figure 14 shows that the high-end

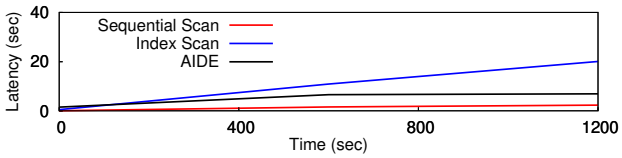


Figure 14: AIDE vs. high-end vanilla PostgreSQL.

vanilla with any access method initially crushes its resource-hungry counterpart (i.e., AIDE). However, the vanilla with the sequential scan gradually slows down as versions pile up, whereas the vanilla with the index scan shows a dramatic increase in query latency due to redundant data access on heap pages. The engine with AIDE benefits from the prescreening effect, thus surprisingly outpacing the high-end vanilla with the index scan dragged out by the growing overhead of version traversal. This result suggests that the vanilla PostgreSQL optimizer must consider the growing version space when choosing the access method.

7 DISCUSSION

We draw some lessons demystifying real-life barriers faced to make good use of intelligent storage for HTAP databases.

Query processing is always resource-hungry. ISP needs a giant jump in expanding its internal bandwidth and access latency to compete with evolving database engines highly optimized for utilizing large storage bandwidth. Contrary to what was expected [15], the gap between the aggregate internal bandwidth and the bandwidth supported by host I/O interfaces is negligible even in the latest high-end NVMe SSDs. So, we can find the benefits of ISP only with queries acting on update-intensive tables mainly due to the inescapable fixed cost of ISP and limited resource capacity.

ISP for HTAP DBMSs costs more than expected. In HTAP, where datasets keep changing, just offloading computation to storage would not work efficiently. To overcome it, database vendors must put substantial efforts into modifying host databases to make recent data and rapid access methods available for storage. In this context, our work epitomizes how much practical effort database developers should put into deploying computational storage in HTAP databases.

8 CONCLUSION

This paper presents AIDE, an intermediate layer aiding MVCC databases for offloading query plans to computational storage, which aims to reduce resource contention and accelerate query operations by eliminating random, redundant data access in storage under massive updates. AIDE transforms all or part of vendor-specific query plans into vendor-neutral formats with version indexes holding canonical tuple identifiers for all visible data in target tables. AIDE prescreens version indexes to filter out invisible data versions, which also precludes version traversal logic from ISP. Our experiments show that vendor-neutral computation with prescreening can make query offloading feasible in both PostgreSQL and MyRocks and boost query processing significantly under HTAP workloads. Although our work has room for improvement, we hope database designers find our approach helpful in deploying computational storage for HTAP DBMSs.

ACKNOWLEDGMENTS

We thank the reviewers for their thoughtful comments. This work was supported by the National Research Foundation of Korea grant funded by the Korean government (MSIT) (No. 2022R1A2C2008427). This work was also partially supported by the Institute of Information Communications Technology Planning & Evaluation (IITP) funded by the Korean government (MSIT) (No. 2021-0-00590).

REFERENCES

[1] 2020. sysbench-1.0.20. Available at https://github.com/akopytov/sysbench.

[2] 2022. HammerDB Version 4.4. Available at https://github.com/TPC-Council/HammerDB/releases/tag/v4.4.

[3] 2022. NTT OSS Center DBMS Development and Support Team: pg_hint_plan-1.4. Available at https://github.com/ossc-db/pg_hint_plan.

[4] 2022. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393): Vitis Analyzer. Available at https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Using-the-Vitis-Analyzer.

[5] Amazon Web Services, Inc. 2022. What Is AWS Glue? https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.

[6] Oracle Corporation and/or its affiliates. 2022. MySQL 8.0 Reference Manual: 15.3 InnoDB Multi-Versioning. https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html

[7] Oracle Corporation and/or its affiliates. 2022. Oracle Database Concept: 9 Data Concurrency and Consistency. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-concurrency-and-consistency.html

[8] Philip A. Bernstein and Nathan Goodman. 1982. Concurrency Control Algorithms for Multiversion Database Systems. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Canada) *(PODC '82)*. Association for Computing Machinery, New York, NY, USA, 209–215. https://doi.org/10.1145/800220.806699

[9] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[10] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 29–41.

[11] Citus Data. 2020. Citusdata: Tools for running CH-benCHmark with HammerDB. https://github.com/citusdata/ch-benchmark.

[12] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1988842.1988850

[13] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data* (Xi'an, Shaanxi, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2490–2502. https://doi.org/10.1145/3448016.3457551

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[15] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[16] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (Jan. 2012), 45–51. https://doi.org/10.1145/2094114.2094126

[17] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 325–336.

[18] Carnegie Mellon University Database Group. 2020. Peloton: The Self-driving Database Management System. https://pelotondb.io/

[19] Carnegie Mellon University Database Group. 2020. Terrier: The Self-driving Database Management System. https://github.com/cmu-db/terrier

[20] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 511–524.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[22] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind Arvind, and Sungjin Lee. 2020. *PinK: High-Speed in-Storage Key-Value Store with Bounded Tails*. USENIX Association, USA.

[23] MemSQL Inc. 2022. MemSQL. https://www.memsql.com/

[24] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.* 9, 12 (aug 2016), 924–935. https://doi.org/10.14778/2994509.2994512

[25] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[26] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 938–950. https://doi.org/10.1145/3448016.3452783

[27] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 49–64. https://doi.org/10.1145/3514221.3526135

[28] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[30] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 534–546. https://doi.org/10.14778/3436905.3436913

[31] Shengwen Liang, Ying Wang, Cheng Liu, Huawei Li, and Xiaowei Li. 2019. InS-DLA: An In-SSD Deep Learning Accelerator for Near-Data Processing. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 173–179. https://doi.org/10.1109/FPL.2019.00035

[32] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. *Greenplum: A Hybrid Database for Transactional and Analytical Workloads*. Association for Computing Machinery, New York, NY, USA, 2530–2542. https://doi.org/10.1145/3448016.3457562

[33] Microsoft. 2022. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/

[34] NuoDB. 2022. NuoDB. https://nuodb.com/

[35] Christos H. Papadimitriou and Paris C. Kanellakis. 1982. On Concurrency Control by Multiple Versions. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, California) *(PODS '82)*. Association for Computing Machinery, New York, NY, USA, 76–82. https://doi.org/10.1145/588111.588125

[36] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. *The Benefits of General-Purpose on-NIC Memory*. Association for Computing Machinery, New York, NY, USA, 1130–1147. https://doi.org/10.1145/3503222.3507711

[37] D. P. Reed. 1978. *Naming and Synchronization in a Decentralized Computer System*. Technical Report. USA.

[38] Erik Riedel, Christos Faloutsos, and David Nagle. 2000. *Active disk architecture for databases*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[39] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (Aug. 2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[40] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 740–755. https://doi.org/10.1145/3477132.3483555

[41] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 731–742. https://doi.org/10.1145/2213836.2213946

[42] Malcolm Singh and Ben Leonhardi. 2011. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for*

Advanced Studies on Collaborative Research. 385–386.

[43] The PostgreSQL Global Development Group. 2022. PostgreSQL: Documentation for PostgreSQL 12: Chapter 15. Parallel Query. https://www.postgresql.org/docs/12/parallel-query.html.

[44] The PostgreSQL Global Development Group. 2022. PostgreSQL: Documentation for PostgreSQL 12: Chapter 29.3. Asynchronous Commit. https://www.postgresql.org/docs/12/wal-async-commit.html.

[45] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active Flash: Towards Energy-Efficient, in-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'13)*. USENIX Association, USA, 119–132.

[46] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. 2020. NKV: Near-Data Processing with KV-Stores on Native Computational Storage. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3399666.3399934

[47] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proc. VLDB Endow.* 15, 10 (sep 2022), 1991–2004. https://doi.org/10.14778/3547305.3547307

[48] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2021. Evaluating List Intersection on SSDs for Parallel I/O Skipping. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1823–1828. https://doi.org/10.1109/ICDE51399.2021.00161

[49] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD In-Storage Computing for List Intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2933349.2933353

[50] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. 2019. Column-oriented database acceleration using FPGAs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 686–697.

[51] Ronald Weiss. 2012. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores* (2012).

[52] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[53] Xilinx. 2021. *SmartSSD Computational Storage Drive*. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html

[54] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 386–399. https://doi.org/10.1109/MICRO50266.2020.00041

[55] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 386–399.

[56] Shuotao Xu, Sungjin Lee, Sang Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. BlueCache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.* 10 (2016), 301–312.

[57] Haichang Yang, Zhaoshi Li, Jiawei Wang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. HeteroKV: A Scalable Line-rate Key-Value Store on Heterogeneous CPU-FPGA Platforms. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 834–837. https://doi.org/10.23919/DATE51398.2021.9474088

[58] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3313–3325. https://doi.org/10.14778/3415478.3415553