GPU-accelerated data management under the test of time

Aunn Raza* Periklis Chrysogelos* Panagiotis Sioulas* Vladimir Indjic*
Angelos Christos Anadiotis* Anastasia Ailamaki* †
*École Polytechnique Fédérale de Lausanne †RAW Labs SA *{first-name.last-name}@epfl.ch

ABSTRACT

GPUs are becoming increasingly popular in large scale data center installations due to their strong, embarrassingly parallel, processing capabilities. Data management systems are riding the wave by using GPUs to accelerate query execution, mainly for analytical workloads. However, this acceleration comes at the price of a slow interconnect which imposes strong restrictions in bandwidth and latency when bringing data from the main memory to the GPU for processing. The related research in data management systems mostly relies on *late materialization* and *data sharing* to mitigate the overheads introduced by slow interconnects even in the standard CPU processing case. Finally, workload trends move beyond analytical to *fresh* data processing, typically referred to as Hybrid Transactional and Analytical Processing (HTAP).

Therefore, we experience an evolution in three different axes: interconnect technology, GPU architecture, and workload characteristics. In this paper, we break the evolution of the technological landscape into steps and we study the applicability and performance of late materialization and data sharing in each one of them. We demonstrate that the standard PCIe interconnect substantially limits the performance of state-of-the-art GPUs and we propose a hybrid materialization approach which combines eager with lazy data transfers. Further, we show that the wide gap between GPU and PCIe throughput can be bridged through efficient data sharing techniques. Finally, we provide an H^2TAP system design which removes software-level interference and we show that the interference in the memory bus is minimal, allowing data transfer optimizations as in OLAP workloads.

1. INTRODUCTION

GPUs have become an increasingly mainstream processing unit for data management systems due to their parallel processing capabilities and their high bandwidth memory which fit the requirements of several, mainly analytical, workloads. However, high performance processing comes with the overhead of data transfers over an interconnect which becomes a bottleneck and results into degradation of the overall query execution time. Accordingly, a large portion of the research in GPUs for data management systems focuses on techniques to mask the overheads coming from the interconnect and maximize the utilization of the GPUs.

The memory interconnect is also a bottleneck in traditional data analytics executed in the CPU and it is mitigated by several techniques. We focus on late materialization and data transfer sharing and we adjust them to the GPU context. Late materialization is a standard technique that brings data to the GPU only when they are needed, usually after the application of filters and selective joins, as opposed to the early approach where data are transferred eagerly before the GPU execution starts. Data transfer sharing adapts the traditional scan sharing techniques to the GPU memory hierarchy in order to amortize the transfer costs from the low-level main memory to the higher-level GPU device memory across queries. To the best of our knowledge, data transfer sharing has not been used in GPUs to date. Both techniques overcome the bandwidth and latency bottleneck of the interconnect by transferring data only when it is necessary and when they satisfy the most possible queries.

The progress in GPU data processing is not limited to the software, but also expands to the hardware. Since the first papers demonstrating GPU utility for data management systems, the architecture of GPUs has evolved rendering them faster and with more memory, which allows them to keep more data locally, thereby boosting query execution. Moreover, NVIDIA, which is one of the major GPU vendors, has recently set restrictions to the use of consumer-grade GPUs, like GeForce, to data center installations [20]. Therefore, there is a transition in the architecture of GPUs which renders the GPU execution faster.

Interconnects have also evolved with some of the most popular standards being PCI express (PCIe) and NVLink. In particular, the first version of PCIe was introduced in 2003 and evolved slowly until PCIe version 4 was introduced in 2017. A new version follows two years later, showing the acute interest in fighting the interconnect bottleneck from the hardware side. Each version of PCIe and NVLink increases the performance and can, potentially, eliminate overheads met in the past and introduce new ones. Nevertheless, a data management system should be able to work transparently regardless the architecture of the GPU and the interconnect.

GPU-accelerated data management systems are mostly evaluated against Online Analytical Processing (OLAP) workloads. Despite their wide applicability, these workloads also guarantee that the underlying data are immutable. Hybrid Transactional and Analytical Processing (HTAP) workloads instead, consider data to be updated regularly and the analytical processing part of the system has to process data coming from transactions, while maintaining consistency and minimizing interference. Therefore, HTAP systems introduce further challenges and they are in the research spotlight, affecting the design of data management systems.

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.

HTAP workloads affect mainly the selection of late vs early materialization approach in the data transfers. Currently, in HTAP systems, a transactional engine either generates a snapshot and sends it eagerly to the analytical engine [17] or it updates data lazily using copy-on-write [15]. Therefore, the selection of the HTAP system design is expected to have a direct impact on the selection of the materialization approach, where there is no clear winner. Instead, in this paper, we show that by eliminating the software-level interference and by separating out the transactional from the analytical query execution, using CPUs and GPUs respectively, the performance of both engines involved, is minimally affected.

This paper evaluates late materialization and data transfer sharing techniques in the face of the evolution on the hardware and on the workload side and it provides insights to data management systems developers and administrators on the interplay among GPU architecture – interconnect – workload. We follow the hardware evolution by using three different hardware configurations that provide different combinations of GPU architectures and interconnects. We follow the workload evolution and evaluate heterogeneous HTAP system design by isolating task-parallel OLTP and data-parallel OLAP workload on CPUs and GPUs, respectively. Our contributions are summarized as follows:

- We show that data management workloads stress the interconnect at a point where even high-end GPU architectures do not meet the expected performance.
- We extend a state-of-the-art GPU analytical engine with a novel hybrid tuple reconstruction strategy that mixes early and late materialization and adapts the concept of scan sharing to the GPU memory hierarchy.
- We demonstrate that, despite the interconnect bottleneck, late materialization and data transfer sharing increase the effective bandwidth, allowing us to take advantage of highly selective queries. This effect is horizontal across the different generations of hardware and the different workloads.
- We show that using a multi-tiered, multi-versioned, storage software interference in the performance of hybrid workloads in heterogeneous HTAP systems is eliminated and we evaluate the impact of interconnect optimizations on the performance of hybrid workloads.

2. BACKGROUND

In recent years, there has been increased interest on GPU accelerated DBMS because of the computational throughput, the high bandwidth memory and programmability that modern GPUs offer. In this section, we provide an overview of the design of systems that perform GPU-based analytical processing as well as heterogeneous HTAP.

2.1 GPU-accelerated Analytical Processing

Analytical operations have been acknowledged as a target for acceleration since the early days of General-Purpose computing on Graphics Processing Units (GPGPU) due to their data-parallel nature. Since then, analytics on GPUs have received significant traction, and related research has spanned by a broad field of work. A multitude of GPU-accelerated analytical DBMS are available, both academic and commercial. We describe the design principles of the state-of-the-art in such systems.

Existing GPU-accelerated DBMS tightly couple their design principles with the topology and hardware of CPU-GPU servers that contain multiple CPU and multiple GPUs. An interconnect, such as PCIe and NVLink, connects GPUs to main memory, enabling data transfers from and to GPU memory. Compared to the main memory of the server, GPU memory has limited capacity and can only store up to medium-sized databases. For this reason, DBMS store the database in the main memory, commonly in columnar layout, and transfer the data on-demand to the GPU through the interconnect. The interconnect has much lower bandwidth than GPU memory and often becomes the performance bottleneck. The bandwidth discrepancy between the interconnect and the GPU is a driving force in the hardware-software co-design of GPU-accelerated DBMS, especially for complex analytical operators such as sorting and joins.

The execution model of GPU-accelerated DBMS considers two factors, the programming model of GPUs and data transfers. GPUs process through a series of data-parallel operations called kernels. The first generation of GPU-accelerated DBMS uses an operator-at-a-time execution model. They map each operator of the query plan to a kernel and they execute the plan bottom-up. The operators consume the complete intermediate input and produce the complete intermediate output before the DBMS executes their parent. In many cases, the intermediate tuples cannot fit in the limited GPU memory and are transferred to the main memory, incurring a high overhead. To fit the intermediate results in GPU memory and to avoid unnecessary transfers, subsequent generations of GPU-accelerated systems follow a vector-at-a-time execution model, based on which the GPU processes data in vectors and pipelines the partial intermediate results through the query plan. Despite decreasing data movement, the vector-at-a-time execution model wastes GPU memory bandwidth by materializing intermediate results in-between kernels and risks making GPU processing the new bottleneck. To further eliminate inter-operator intermediate results, state-of-the-art DBMS compile sequences of operators into unified data-parallel kernels. Both research [5, 6] and commercial [18] systems use just-in-time code generation to improve query performance.

Modern GPU-accelerated systems also overlap execution with data transfers to reduce the total execution time of queries. One approach is to access the main memory directly from the kernels, over the interconnect, by taking advantage of the unified address space for the main memory and the GPU memory. Another approach, which is compatible with the vector-at-a-time execution model, is to use asynchronous data transfers to pipeline the use of the interconnect and the GPU. The GPU processes a vector of data, while the transfer of the next vector is on-the-fly. Overlapping transfers and relational processing hides a major portion of the time spent on GPU execution, making the interconnect bottleneck even more profound. We advocate for a data transfer-centric GPU-accelerated engine design that combines reduced pressure to the interconnect and an analytical throughput that exceed the interconnect capacity.

2.2 HTAP on heterogeneous hardware.

Data analytics have long considered that the underlying dataset remains immutable. However, business analytics requirements have lately raised the bar by putting freshness as an additional property that OLAP systems should satisfy. The demand for freshness led to the introduction of Hybrid Transactional and Analytical Processing (HTAP) systems, which combine both transactional and analytical functionality. Most commercial systems including, but not limited to, DB2, SQL Server/Hekaton, SAP HANA and Oracle DB are examples of HTAP systems. HTAP focuses on the *freshness* aspect of the data in par with the performance of transactions and analytics by providing *fresh* data efficiently from the transactional to the analytical part of the system.

The primary concern for HTAP systems is the interference between two logically (or physically) separated engines working on the same dataset, with one of them updating it. A significant factor that determines the interference is the mechanism for propagating updates from the transactional to the analytical engine. The choice of a mechanism spans a broad spectrum of HTAP architectures. On one extreme of the spectrum, HTAP architectures co-locate the engines in a single system and use an efficient snapshotting mechanism to mitigate interference. [14] is a typical example following this approach. These systems rely on hardware-assisted copy-on-write (COW) to separate read-only from read-write data. The transactional engine writes on the new page, transparently, since the new page has the same virtual address for the application while the old page is moved to the page table of the analytical engine and again maintains the old virtual address. On the other extreme of the spectrum, systems such as [17] follow an HTAP architecture that executes queries in batches. In this case, the transactional and the analytical engines run in isolation in different nodes of the system. Upon the end of an epoch, the transactional engine extracts a delta log and sends it to the analytical engine. The analytical engine applies the log to its storage and then executes the batch of queries that have been waiting for their execution. Compared with the previous, COW-based, architecture, this one opts for isolation, whereas the other opts for freshness since it takes a snapshot every time an analytical query session is about to start.

The adoption of GPUs for analytical processing has sparked interest in Heterogeneous HTAP (H^2TAP) [3]. Heterogeneous HTAP uses hardware and workload heterogeneity in a synergetic fashion. It assigns the latency-critical transactional workload to one or more CPU nodes and the bandwidth-critical analytical workload to GPUs. Caldera [3] is a prototype H^2TAP that performs HTAP over heterogeneous hardware using COW software snapshotting. Caldera demonstrates that COW introduces overheads because it stresses the memory bandwidth in two directions: copying the data inside the DRAM for the write operations and copying the data from DRAM to the GPU for the analytical processing. We study the transactional-analytical interference in the H^2TAP context and in conjunction with the utilization of the interconnect. We eliminate COW and we use a two-tiered, multi-versioned storage to send fresh data from the DRAM to the GPU for analytical processing, moving the focus from the software to the hardware-level interference, especially at the interconnect level, in H^2TAP.

3. THE INTERCONNECT BOTTLENECK

H^{2}TAP systems bridge the GPU-accelerated analytical engine with the transactional storage via the interconnect. The interconnect is the central component of H^{2}TAP and the most critical factor in the performance profile of the two constituent engines. First, the interconnect bandwidth imposes a hard limit to analytical throughput, because the engine needs to transfer to-be-scanned columns to the GPU for any analytical processing. Second, interconnect utilization consumes memory bandwidth and therefore serves as a metric for resource sharing between the transactional and the analytical engine.

Figure 1 presents the impact of interconnect in analytical query processing by showing execution times of query family 2 from Star Schema Benchmark across combination of consumer- and server-grade GPUs with slow and fast interconnects.

The bars represent the execution time for executing queries Q2.1, Q2.2 and Q2.3, which have the same query template and decreasing selectivities from 1 to 3, with combination of server-grade GPUs with PCIe and NVLink, and, consumer-grade GPU with PCIe. The dashed lines represent the time required for transferring the working set of the queries (9.2 GB) to the GPUs using different interconnect technologies, specifically PCIe (black) and NVLink (green). Based on this experiment, we observe the following properties hold for GPU-accelerated data management systems:

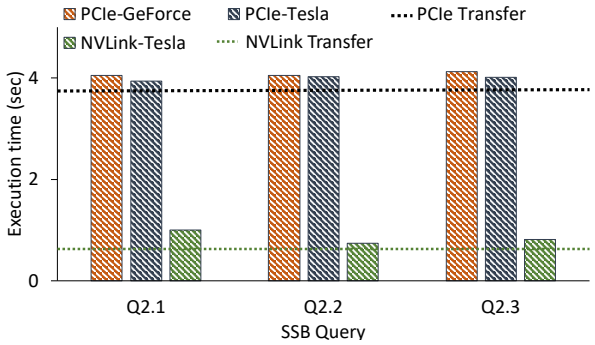


Figure 1: GPU query execution on CPU-resident SSB (SF=1000) across different GPU architectures & interconnects

- **GPU Underutilization.** GPU-accelerated query processing is interconnect-bound. Figure 1 shows that the execution time for all queries is at most 11% higher than the transfer time that the interconnect requires for the given configuration. For PCIe-based configurations, upgrading the GPUs from consumer-grade GTX 1080 (320 GB/sec) to server grade V100 (900 GB/sec) brings a marginal performance improvement of $0.5-2.7\%$. By contrast, upgrading the interconnect from PCIe to NVLink in a V100-based configuration yields a speedup of $3.94-5.42$. The interconnect bandwidth dictates the query execution time, whereas the processing capabilities of the GPU are underutilized, especially for slower interconnects. Recent work shows that even expensive queries are interconnect-bound when PCIe is used [24, 8, 9]

- **Selectivity-Insensitive Performance.** The size of the working set determines the execution time of the query. Figure 1 shows that the execution time of the query is roughly constant, even though the selectivity is decreasing. The observation is counter-intuitive. Users expect queries with more selective filters to have a shorter time to result. The expectation holds for CPU-based analytical systems which access main-memory directly at fine granularity, but not for GPU-accelerated systems that eagerly transfer the whole working set of the query over the interconnect, one vector-at-a-time.

These two properties of GPU-accelerated OLAP engines are undesirable. It indicates that the analytical engine fails to fully take the advantage of hardware and workload characteristics in the execution environment.

In H^{2}TAP systems, the shared main-memory bandwidth is also a source of performance degradation of either, analytical or transactional throughput, or both: OLTP reads and updates at tuple-granularity while OLAP reads at column-granularity. This delegates to the underlying memory sub-system – CPU DMA controllers – to schedule and prioritize incoming requests. In general, sequential scans consume full memory bandwidth while starving small, random-access requests. Moreover, not only at hardware level, but also, HTAP systems presents a challenge of data-freshness, that is, OLAP to access data updated by OLTP engine with snapshot isolation guarantees.

In Section 5, we discuss modifications to the execution paradigm that enable our engine to overcome these challenges by reducing pressure over interconnect and main-memory.

4. AN H^{2}TAP ENGINE

In this section, we provide an overview of our H^2TAP prototype engine that combines an analytical GPU-accelerated engine with a CPU-based transactional engine. First, we discuss the design principles of the transactional and the analytical components of our system. Then, we discuss the interface between the two that enables HTAP.

4.1 GPU-accelerated Analytical Engine

The analytical component of our H^2TAP prototype is Proteus [8, 9], a state-of-the-art GPU-accelerated analytical engine. Proteus targets multi-CPU multi-GPU servers by exploiting the heterogeneous parallelism across all processing units. However, we focus on execution configurations that only use the GPUs. By using data and control-flow operators, it combines state-of-the-art scalability and performance with data and device-agnostic operators.

Proteus is a columnar DBMS that uses the main memory for storage. To process queries using the GPUs, it transmits blocks of data over the interconnect using asynchronous transfers and subsequently consumes it using a vector-at-a-time execution model. It schedules execution and transfers using CUDA streams. To increase the inter-operator efficiency and eliminate the materialization of intermediate results, Proteus uses just-in-time code generation to fuse sequences of pipelined operators into unified kernels using LLVM. Concerning data transfers and the execution model, Proteus incorporates all the latest optimizations of state-of-the-art systems. In Section 5, we extend Proteus with interconnect-related optimizations, specifically optimizing data access method per-query and sharing data across queries.

4.2 CPU-based Transactional Engine

On the transactional side, we use a two-tiered, multi-version storage in order to separate out effects of resource interference at hardware and software level. The first tier is a two-versioned storage similar to the Twin Blocks [7, 19]. During every epoch, only one version is updated and the other is shared with the analytical engine. The epoch can either change periodically or through explicit requests. The second tier is a multi-versioned storage used by the transactions. Every transaction commit is written to the first tier update-reserved version, in order to be immediately available to the analytical side upon the change of the epoch.

4.3 Heterogeneous HTAP

Proteus is extended to be a complete HTAP system by introducing snapshot plugins, which are interfaces from OLTP engine to OLAP engine for getting data pointers to fresh transactional snapshots. Snapshots are triggered either on-demand or periodically on epochs, while maintaining the bounded staleness on transactionally inactive versions. This allows the two engines to be isolated at software level by removing high-latency copy-on-write operations for OLTP or OLAP traversing version-chains at tuple-level. With two-tiered storage manager in OLTP engine, Proteus can access one analytical snapshot-at-a-time and can execute all analytical queries in a batch, similar to [17]. However, in case where the system needs to maintain snapshots of variable freshness, this approach can be extended by updating OLTP storage manager from two-tiered to n-tiered, having a circular buffer of consistent data snapshots.

At hardware level, the two, analytical and transactional engines are isolated by explicit scheduling of compute resources. Task-parallel OLTP workload is executed on CPUs while data-parallel analytics on GPUs.

5. TRANSFER OPTIMIZATIONS

This section first describes the two approaches that we study in this paper for mitigating the GPU overheads, namely lazy data transfers and data transfer sharing.

5.1 Lazy transfers

Although the cost of transferring data is high, GPU-accelerated DBMS eagerly transfer data to GPUs before processing. Prefetching blocks of data has multiple advantages. First, it guarantees to the subsequent operations that the data are in an accessible memory, as depending on the hardware configuration, a GPU may not have direct access for reading data in any other memory location in the system. Second, the access pattern over the interconnect is sequential, and thus, it allows full utilization of the available bandwidth. Third, kernels access data stored in GPU memory and benefit from the high memory bandwidth and low memory latency as a result. Fourth, the copy of the data in the GPU memory can be used multiple times within the lifetime of a query. However, despite the benefits of asynchronously prefetching data to the GPU, the approach tends to over-fetch by transferring data that is not required. Query performance becomes insensitive to selectivity.

Modern GPUs have a unified address space and allow on-demand access to data in the CPU memory. The mechanism constitutes a lazy way of transferring data by pulling into the GPU and can bring significant benefits to GPU query processing. Primarily, it reduces the amount of transferred data for highly selective queries by leveraging the higher granularity of interconnect accesses. The kernels load columns accessed after the evaluation of predicates only partially. Furthermore, the memory footprint of query processing decreases because intermediate buffers are unnecessary. Finally, lazy accesses allow optimizations such as compression and invisible joins. The two data transfer methods correspond to the tuple reconstruction policies. Eager prefetching corresponds to early materialization, while lazy transfers correspond to late materialization. The two approaches define a trade-off between the utilization of GPU and interconnect hardware and the decrease of data transfer requirements.

We extend our system to support a hybrid transfer strategy that enforces a transfer policy at the column level. We route, transfer and process data in blocks, which are logical horizontal partitions. A data transfer operator copies the eagerly transferred columns of the block, or any columns inaccessible from the GPU, to GPU memory and forwards a pinned main memory pointer for the lazily transferred columns. Then, during execution, the GPU transparently reads the former from the GPU memory and the latter over the interconnect using the UVA. With the proposed design, we achieve more complex reconstruction policies that deliver better performance than both pure eager and pure lazy transfer strategies because we optimize for each column judiciously, without adding extra complexity to the query engine design.

5.2 Transfer sharing

Data movement from the main memory to the GPU constitutes the main bottleneck for query processing due to the bandwidth disparity between the GPU device memory and the interconnect, for all interconnect technologies. Transfers over the interconnect are in the critical path of the execution. Thus, as long as no transfers are redundant, we cannot decrease the latency of GPU-accelerated query processing further. However, in an environment of concurrent query execution, we can reduce the overall data movement

for query processing by sharing transferred data across multiple queries. As a result, we can amortize the cost of transfers over the interconnect and increase the throughput of the DBMS. The described technique mirrors the concept of scan sharing from disk-based DBMS but adjusted to the GPU memory hierarchy.

To achieve transfer sharing, we implement a buffer pool inside the high bandwidth memory of each GPU. We model transfers as mem-move operators [9] and for each such operator, we match the subtrees across queries to identify sharing opportunities. Unlike scan-sharing, it is possible to share intermediate results of the query plan as well, for example, when shuffling data between multiple GPUs. When a query demands a block of data for processing, it registers the request in the buffer pool of the respective GPU. If a request already exists for the block in question, the consumer operators synchronize to wait until the transfer is complete. Otherwise, the system initiates a new transfer to an empty buffer and registers the request in the buffer pool. The design is flexible both in terms of the content of shared data and policies of the buffer pool regarding evictions, pinning, or priorities. In our current implementation, we assume query batches and pin buffers to ensure maximum sharing and evict the most-recently unpinned buffer. This design allows encapsulating sharing in the buffer pool management and the transfer operator in a non-intrusive manner.

5.3 Global Transfer Plan

We present two techniques for improving the utilization of the interconnect. On the one hand, transfer sharing amortizes the cost of data transfers across queries. On the other hand, the hybrid transfer strategy optimizes transfers within each query by maintaining a balance between reducing the amount of data accessed and increasing the interconnect and GPU efficiency. The two techniques can co-exist, but they are not orthogonal. Lazy transfers occur directly over the interconnect and thus preclude any form of sharing. However, sharing is still possible for the prefetched columns.

In this context, the transfer policy is a cross-query optimization problem. The query optimizer needs to decide which columns to pull lazily and which columns to push eagerly, and express the decision on the physical query plan. The optimizer decides to share eagerly pushed data by default because transfer sharing incurs no extra effort. Also, it decides against using lazy transfers if a copy of the data is available in the buffer pool, due to other concurrent queries. In that case, sharing is again preferable. To determine the transfer that should be lazy, the optimizer examines the access pattern for the column in question, the aggregate amount of required data across queries, and a cost model that represents the impact of interconnect latency. The decision takes the observed execution time into account, which is the maximum of the GPU execution time and the required time for transfers over the interconnect.

6. EVALUATION

This section includes the results of our experimental evaluation. First, we describe the hardware that we used to execute our experiments, and the benchmarks that we used to derive our workload, the software used and the undertaken evaluation methodology.

Then, we present the experimental results of the proposed data-transfer techniques per query and transfer-sharing approach across queries, to show the corresponding benefits and trade-offs with each approach. Finally, we show the performance interference in HTAP workloads combined with different data-transfer techniques.

6.1 Hardware and Software

Hardware. To study the effects across consumer- and server-grade GPUs connected over relatively slow and fast interconnects, we executed our experiments on three different servers. The first and the second server have 2 x 12-core Intel Xeon Gold 5118 CPU (Skylake) clocked at 2.30-GHz with HyperThreads, that is, 48 logical threads and a total of 376-GB of DRAM. The first server has 2 x NVIDIA GeForce 1080 GPUs (consumer-grade), whereas the second one has 2 x NVIDIA Tesla V100 GPUs (server-grade), over PCIe v3 in both cases. The third server is based on IBM POWER9 equipped with 2x16-core SMT-4 CPUs clocked at 2.6-GHz, a total of 128 logical threads and 512-GB of DRAM. Like the second server, it has 4 x NVIDIA Tesla v100 GPUs, but they are connected to CPUs over NVLink 2.0 interconnect with 3-links per GPU. In all the experiments, we use 2 GPUs, one local to each CPU socket, even in the 4-GPU server. In the rest of the section, we will refer to these servers as `PG` for the first one, `PT` for the second one and `NT` for the third one. We use the first letter, `P` and `N`, to signify the interconnect standard (PCIe/NVLink) and the second, `G` and `T`, for the GPU architecture (GeForce/Tesla). Currently, there is no consumer-grade GPU with support for NVLink.

We use the three hardware configurations to show how the evolution of GPU architectures and interconnects affect the performance of the access methods and data transfer sharing. By comparing PG with PT, where we change only the GPU architecture, we observe how the increased GPU capabilities affect performance. Then, by comparing PG with NT, we keep the same GPUs, but we change the interconnect from PCIe to NVLink, thereby evaluating the effect of the interconnect to the query performance. Migration from Intel to POWER CPU architecture was necessary, since NVLink is available as a CPU-GPU interconnect only on IBM POWER servers, to the best of our knowledge.

Software. We use in-house, open-source software for our evaluation. OLAP queries are executed on Proteus, a state-of-the-art query engine which executes queries on both CPUs and GPUs and relies on code-generation [9, 8]. We extended Proteus by adding CPU-only OLTP engine, named Aeolus. Aeolus is configured with a multi-versioned storage manager and uses a columnar storage layout, whereas its transaction manager uses MV2PL with deadlock avoidance for concurrency control. The snapshot manager of Aeolus, pre-faults memory chunks sizing to main-data size in order to provide fresh snapshots to OLAP.

For the scope of this study, we execute all OLTP workloads on CPUs. For OLAP, all experiments use 2-GPUs, one per NUMA node and GPU-Only execution mode of Proteus. For explicit resource allocation, OLAP reserves 1-physical core per socket (4-logical threads in PG & PT, and 8-logical threads in NT) for final data reduction and management threads.

Benchmarks. We evaluate our methods using two benchmarks. For OLAP-only experiments we use the Star Schema Benchmark (SSB) [21] with scale factor 100. HTAP experiments are performed using the CH-benchmark [10] which combines two industry standard benchmarks, TPC-C and TPC-H, for transactional and analytical processing systems, respectively. We use the schema as defined by the benchmark, which inherits the relations specified in TPC-C and adds three more relations specified in TPC-H, which are `Supplier`, `Nation` and `Region`. To better study the HTAP effects, we scale the database size following the TPC-H approach by a scale factor SF. Accordingly, the size of the `LineItem` table becomes $SF * 6,001,215$. We fix 15 `OrderLines` per `Order` when initializing the database and we scale the number of records in `OrderLine` to $SF * 6,001,215$. In contrast to TPC-H, and as per TPC-C specification, every `NewOrder` transaction generates five to ten order lines per order. Unless stated otherwise, all experiments are conducted on initial database with scale factor 100. For the transactional workload, we assign one warehouse to each available

worker thread, which generates and executes transactions simulating complete transactional queue. As the CH benchmark does not specify selectivities for conditions on dates, we select values for 100% selectivity, which is the worst case for join and groupby operations. To study the interference between the hybrid workloads, all experiments begin by acquiring the snapshot of the initial transactional data, in order to keep analytical working set constant across queries while transaction executes in parallel.

6.2 Reducing transfers

To evaluate data access methods per-query and transfer sharing across queries, we use Star Schema Benchmark with a scale factor (SF) 100 and pre-load the data on the CPU side. We use a columnar layout and each of the queries of the first three flights has a working set of 9.2GB, while queries of the last group have a working set of 13.8GB. Prefetching and explicit bulk copies over the interconnect happen in the granularity of blocks (2MB huge pages in our system) and transfers overlap with execution [9].

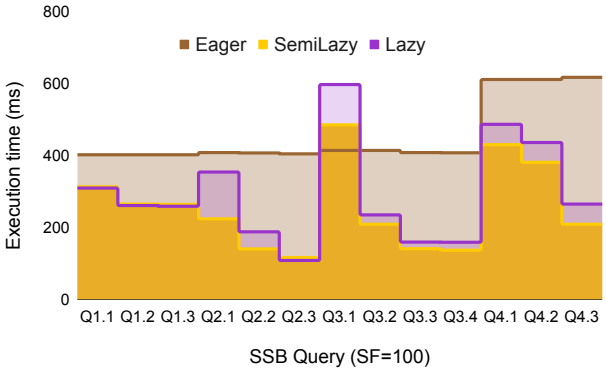
6.2.1 Lazy data transfers

Methodology. This section evaluates the performance of different data transfer techniques across two axes: the interconnect characteristics and the query selectivity. First, we show how the interconnect bandwidth affects query execution and then evaluate the performance gain achieved by lazily accessing the input. Each query fetches the required data over the interconnect and any transfer sharing or caching is disabled, to simulate the case of reading fresh data from the CPU. *Eager* prefetches all the data to the GPU memory. In the *lazy* method, all data are accessed directly from the GPU threads during execution, without an explicit bulk copy. As a result, during kernel execution, GPU threads are experiencing a higher latency during load instructions for input data, compared to the eager method, where data are accessed from the local GPU memory. To reduce requests to remote data, and as a result the actual transferred volume, in the generated code, all read requests are pushed as high into the query plan as possible. For the *SemiLazy* method, we prefetch the firstly accessed column of the fact table for each query and access the rest of the working set using the lazy method. For query groups 2-4 the first column is the foreign key using in the first join, while for query group 1 that filters the fact table before the first join, we prefetch the two columns that are used in the filter predicates.

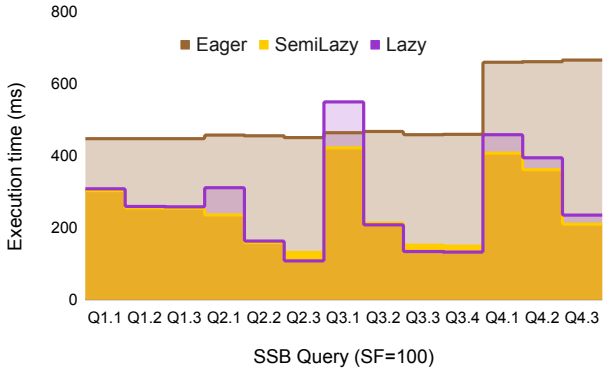
Eager access methods achieve throughput (working set over execution time) very close to the interconnect bandwidth for all three hardware configurations. The only exception is Q3.1 on NT, which has the lowest selectivity and combined with the multiple joins stressing the GPU caches, it causes high memory stalls. The bandwidth of the PCIe interconnect is low enough that these stalls have a minimal impact on the execution time, as they are hidden by the transfer time. In contrast, the ∼5x higher bandwidth of NVLink 2, makes transfer times slower than kernel execution times, causing the stalls to become the new bottleneck. For all other queries, kernel execution is overlapped and hidden by data transfers.

Lazy access methods allow for reducing the transferred volume. SSB queries are highly selective and thus performance improves in almost all queries, except of query 3.1. As Q3.1 is compute-heavy, its performance degrades due to the increased latency that materializes as memory stalls. Inside each query group, the benefit compared with the Eager method increases, as queries become more selective.

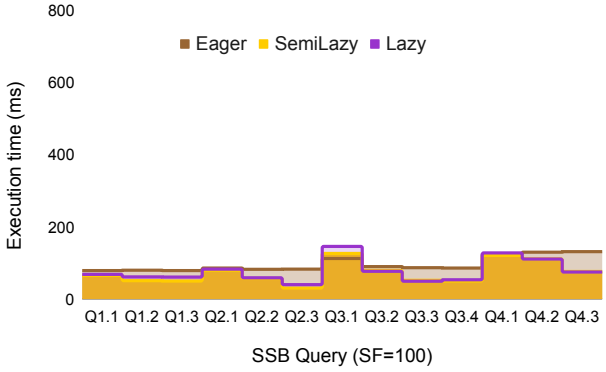
SemiLazy improves upon Lazy by reducing the main penalty for lazy access methods. While laziness reduces the transferred data, it does so in the cost of higher memory latency. Nevertheless, some



(a) PG (consumer-grade GPU & PCIe 3)



(b) PT (server-grade GPU & PCIe 3)



(c) NT (server-grade GPU & NVLink 2.0)

Figure 2: Performance of the different access methods on SSB (SF=100) on the three different configurations.

columns are accessed almost at whole. Prefeching those columns decreases the overhead of laziness at the expense of fetching extra tuples. This approach brings the performance closer to the expected execution times predicted by taking into consideration the selectivity of each individual operation. The higher effect of SemiLazy is on the low-end GPU, where compute resources are limited compared to the server-grade GPUs. Additionally, while Lazy improves mostly the performance of very selective queries, SemiLazy access methods improve queries that are less selective and have multiple dependent operations, like Q2.1. The smaller latency of NVLink, compared to PCIe, reduces the impact of this method.

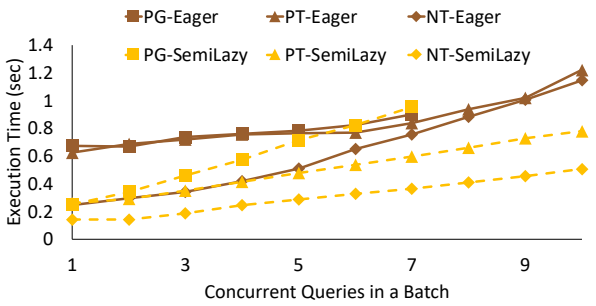


Figure 3: Transfer sharing with eager and semi-lazy transfers

Summary. Eager methods reduces memory stalls during execution and allows greater opportunities for caching & sharing data transferred to the GPUs. Eagerness transfers unnecessary data and penalizes query execution on selective queries. On the other hand, Lazy access methods reduce the execution time of highly selective queries, by avoiding unnecessary transfers, at the expense of memory stalls and data dependencies which are combined with partially transferred columns that impede column reusability. SemiLazy adapts between the two to achieve the best of both worlds, by using a different method for each column: if a query is going to touch most of a column, it will be prefetched eagerly, while columns that are only accessed depending on evaluated conditions are lazily accessed. This results in SemiLazy adapting between the two methods and improving the performance of queries that are selective but also compute-intensive.

6.2.2 Data transfer sharing

Methodology. To evaluate the impact of transfer sharing, we execute queries in batches of variable size. Each batch contains independent instances of the same query (Q4.3 of SSB SF=100). We reserve a buffer pool for each transferred relation and we avoid any other work sharing opportunities, for example by not sharing join hash tables across queries. We report the execution time of the whole batch, from the moment it was submitted until the moment that the last query finishes, and compare the results on the three hardware configurations, PG, PT and NT.

Figure 3 shows the execution time of batches of 1 to 10 instances of the join-heavy query 4.3. In this plot, we observe the impact of transfer sharing Eager and SemiLazy access methods. Full-lazy access methods are incompatible with transfer sharing as they do not materialize results to GPU memory and thus transfers can not be shared. We leave as future work caching lazily fetched tuples in memory and accessing them in following executions. In SemiLazy accesses, we select the same columns as in the experiments of section 6.2.1 for eager prefetching as well as sharing across the queries in the same batch.

The desktop GPU in PG has smaller memory than the server-grade GPUs in PT and NT. The limited memory can fit the data structures, such as hash-tables and pinned input buffers, for up to 7 concurrent queries without using out-of-GPU joins. While our system supports joins without materializing the whole hash-table on the GPU [24], we temporarily disable it and for PG we report up to its limit of 7 queries, to allow for a fare comparison.

Eager transfers in the PCIe-based configurations initially have a marginal increase in execution time because shared transfers mask GPU execution and only happen for the first access to each block of input data. PG and PT have a sharper slope starting at 5 and 6 queries respectively because the corresponding GPUs become execution-bound and are unable to mask outstanding processing. By contrast, NVLink cannot mask execution for any batch size >2 and NT shows increasing execution time throughout. For more than 7 queries, the execution time of NT is only slightly lower than that of PT. Thus, with sufficient sharing the interconnect becomes increasingly irrelevant for performance and the GPU becomes the dominant factor, as transfer sharing reverses the bottleneck. After 8 queries, NT and PT have very similar performance and execution time grows with the same slope.

SemiLazy accesses combined with transfer sharing allows for a lower execution time per-batch for smaller batches, but incurs a higher penalty for each query added to a batch. The execution time increases with the number of queries even from the second query added to a batch, as lazily accessed columns are still accessed over the interconnect in a per-query basis. Nevertheless, for a small number of queries, the avoided traffic reduces the total execution time. Both GPU and interconnect hardware influence performance when the engine combines shared and lazy transfers. SemiLazy-enabled transfer sharing has a higher slope for PG compared to PT and NT due to the kernel scheduling: thread blocks from different kernels that access the same inputs overlap and run concurrently in the server-grade GPUs, due to their launch configuration and the more available resources like Streaming Multiprocessors and registers. Thus, they reuse lazily fetched data as they are accessed. The offset between the PT and NT configuration is due to the interconnect bandwidth, which is used to fetch the lazily fetched data.

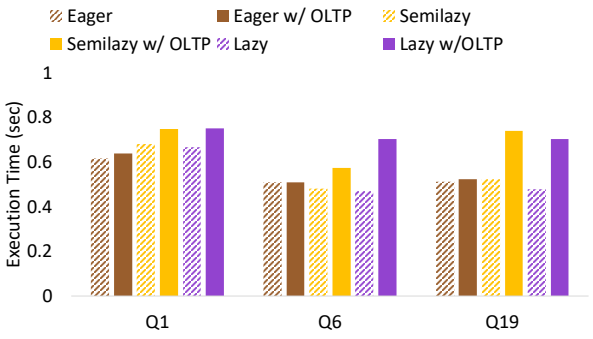
Summary. Data transfer sharing masks interconnect bottleneck up to the point where the workload becomes execution-bound, hence maximizing GPU utilization overall. When combined with access method, the speedup by SemiLazy access methods cascades to sharing opportunities; the query batch shares the eager-scan of the query and fetches only the qualifying tuples over interconnect. Additionally, combining SemiLaziness with sharing allows for packing more queries in to the same batch and improves the performance gains for smaller batches.

6.3 HTAP effect

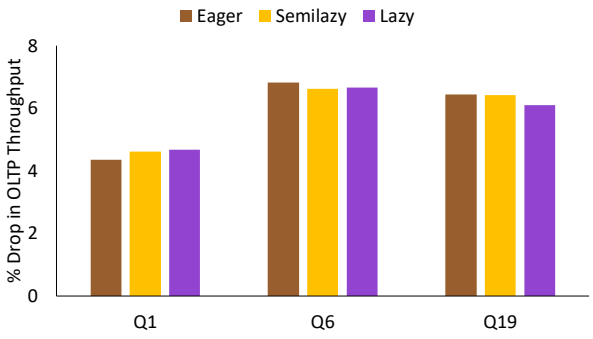
This section evaluates the effect of a hybrid transactional and analytical workload in a heterogeneous HTAP system design, where OLAP execute queries on GPUs while OLTP executes transactions on CPU, simulating full task-parallel workload on CPUs while data-parallel workload on GPUs.

Methodology. To evaluate performance interference in hybrid workloads in the presences of shared main-memory, we use three queries of CH-benchmark, executed on initial database snapshot, sized of TPC-H SF100. Representative operations and working set of each query used for this experiment is presented in 1. Each analytical query is executed as a sequence of five queries on GPUs with concurrent OLTP workload on CPUs. For analytical queries, we measure absolute response times in seconds, while for transactional workload, we measure the percentage drop in transactional throughput during the lifetime of analytical query sequence, relative to the throughput before execution of the analytical query sequence. Each analytical query sequence is executed with different access methods, i.e., eager, semilazy and lazy. and, all experiments are conducted across three-servers; PG, PT and NT.

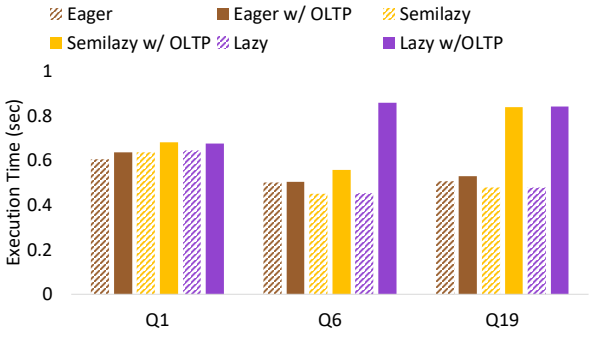
Figure 4a, 4c, and 4e shows the execution times of three analytical queries from the CH-benchmark with different access methods. Figure 4b, 4d, and 4f shows the corresponding relative percentage drop in OLTP throughput while executing analytical queries concurrently in GPUs.



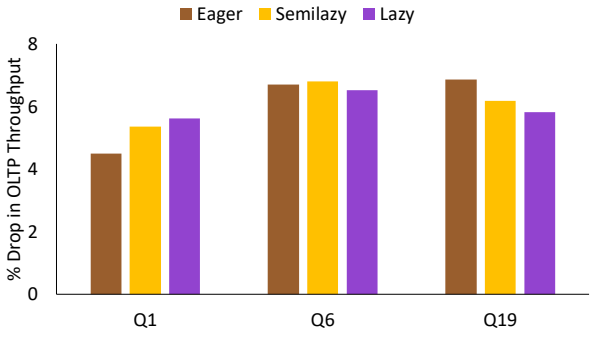
(a) PG - OLAP response times



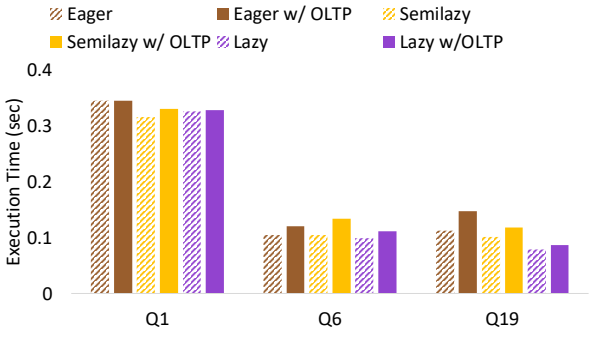
(b) PG - Percentage drop in OLTP performance



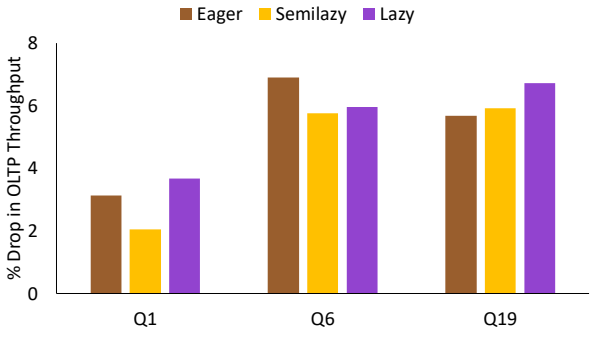
(c) PT - OLAP response times



(d) PT - Percentage drop in OLTP performance



(e) NT - OLAP response times



(f) NT - Percentage drop in OLTP performance

Figure 4: Effects of OLAP access methods on performance interference in hybrid workloads

CH-Query	Query Ops	Working Set
Q1	scan-filter-group-aggregate	13.4GB
Q6	scan-filter-aggregate	11.2GB
Q19	scan-join-filter-agg	11.2GB

Table 1: CH-Queries operations and working-set size

In HTAP workloads, OLTP and OLAP have conflicting memory accesses at different granularities. OLTP reads and updates the memory one tuple at a time, while OLAP reads at column granularity in general. Contrary to CPU-only HTAP systems, by eliminating performance interference at software level, the performance interference in H^{2}TAP is dictated by the interconnect and the access method used over it.

OLAP-only impact of access methods. Q1 and Q6 have 70% selectivity and thus the different access methods produce small savings: most cache lines of the input data are touched. Similarly, Q19 seems small improvements by using the Lazy and SemiLazy access methods due to the low selectiveness of the predicates. In the PG & NT configurations, Q19 is faster using the Lazy approach, instead of the SemiLazy, as our policy for selecting the eagerly transferred columns selects only the column that is used for joining `orderline` with `item` and thus sees benefits from laziness only in accessing the last column, that is used for the reduction after the join. Additionally, as we map one warehouse to each physical core, Q19 is more selective in NT configuration, which explains the relatively bigger time saving, compared to the other two configurations, when using Lazy and SemiLazy access methods.

Interference. As OTLP is latency sensitive and OLAP is bandwidth sensitive, OLTP see only a ∼5% reduction in transactional throughput, due to the hardware isolation.

Eager access methods experience the least performance interference across both GPU architectures and interconnects. In the Eager access method, every block of data that is going to be consumed by a GPU kernel is staged into the GPU memory, before the kernel is executed. Thus, OLAP performance is capped by interconnect bandwidth and the kernel execution times only depend on accesses on local GPU memory. The interconnect transfers leave enough DRAM bandwidth for OLTP to proceed normally: PCIe consumes at max 16% of DRAM bandwidth in PG and PT, while NT can consume up to 50% of DRAM bandwidth. On the other hand, OLTP workload is dominated by random accesses and is latency sensitive, hence is not effected by eager transfers, as sequential scans have minimal pressure on CPU memory DMA controllers.

Lazy access methods cause performance interference in OLAP response times. While laziness reduces the data volume transferred over the interconnect, it also causes more out-of-order requests to the CPU memory, as they are spawned from the different concurrent thread blocks running in the GPU. This causes a higher observed latency, when the CPU memory is loaded serving requests to the OLTP engine. Interference in Q6 & Q9 in configuration PG is smaller compared to PT as there are less thread blocks concurrently active in the GPU, can thus access patterns to CPU memory are more regular.

SemiLazy access methods experiences the same interference as Lazy ones. For Q6, the total relative interference is reduced compared to the Lazy access method, as some columns are accessed eagerly and thus the pattern that causes interference for the Lazy method is now reduced to only some column accesses. For Q19, the same holds but the OLAP engine experiences the similar interference. This is caused by the aforementioned column selection: the columns whose almost all values are accessed are lazily touched, while more selectively accessed columns are eagerly transferred. As a result, SemiLazy for Q19 converges to a Lazy access method.

Summary H^2TAP requires fresh data to be brought from the CPU memory to the GPUs. To reduce the amount of data transferred over the interconnect we use sharing of data transfers to increase reusability and lazy accesses to only fetch data we need. Nevertheless, these two approaches contradict each other. SemiLazy allows reducing the amount of transferred data, while reusing their bigger portion, by prefetching whole columns, whenever most of it will either way be accessed. Though, combining H^2TAP with SemiLazy challenges the traditional understanding of OLAP-OLTP interference: SemiLazy makes GPU accesses to be remote reads on potentially sparse and thus random locations. These locations cause the memory bus to treat both the CPU and the GPU as devices that do random accesses, but now the OLAP engine resides on the remote devices connected over an interconnect. As a result, OLAP, the engine that runs on the remote device pays the most due to the multiple and sparse memory requests that happen from a remote device.

7. DISCUSSION

Our experiments show the effect of the evolution of hardware and workload on GPU-accelerated data management systems. We focused on the interconnect bottleneck, since data analytics workloads are bottlenecked by the memory interconnect, even when they are executed in CPUs. We show that even for high-end interconnects with increased bandwidth and reduced latency many queries are still bound by the transfers. However, this can be attributed to the high-end GPUs that these interconnects come with, which makes them process data fast. On the other hand, we show that if the server is restricted into lower bandwidth setups, we can use lower-end, and typically more power efficient, GPUs to support queries that are bottlenecked by the interconnect. Accordingly, data management specialized GPUs would trade compute power with more interconnect links or device memory.

On the software side, we show that we can increase the effective bandwidth of the interconnect, and thus query performance, by applying late materialization and data transfer sharing over the interconnect. Effectively, late materialization yields more benefits on queries with highly selective filters and joins. Similarly, data transfer sharing opts for cases where multiple queries transfer similar data over the interconnect. Both approaches are generic and therefore beneficial across newer and older interconnects and servers.

Finally, we show that HTAP systems can fully exploit the hardware isolation offered by a system where transactions and analytics are executed on different devices. Specifically, in case of H^2TAP, fresh data are generated on the CPU side and the GPUs have to fetch them to provide fast analytics on fresh data. This causes excessive data transfers and high interference as both subsystems compete for the same resources. To reduce the data transfers and reuse transferred columns, we propose a SemiLazy access method which allows saving on data transfers by taking advantage of the fine-granularity direct memory accesses, while it also increases sharing opportunities. Nevertheless, the random access pattern of SemiLazy increases the number of memory requests and thus the impact of memory latency to data on the CPU memory. Secondly, to reduce interference, we minimize OLAP's access to CPU memory through SemiLazy accesses and transfer sharing. To enable both laziness and sharing, we rely on the Twin Block storage support of the transactional engine that allows the OLAP engine to avoid costly index traversals. In summary, contrary to the common belief, when moving OLAP to accelerators, OLTP can also starve OLAP if there is not enough remaining bandwidth to send data to the accelerators.

8. RELATED WORK

GPU analytical processing. GPU-accelerated DBMS utilize GPUs to accelerate analytical query processing. Earlier systems such as CoGaDB [4] and GDB [13] use an operator-at-a-time model. GPUDB [25] introduces optimizations to overlap computation with data transfers. State-of-the-art GPU-accelerated systems follow a vector-at-a-time execution model to fit intermediate results in GPU memory. Lastly, academic [5, 6, 22, 9] and commercial [18] systems extend vector-at-a-time execution with pipelining and query compilation to eliminate the cost of materializing intermediate results altogether.

Parallel query execution. Modern servers are typically multi-socket multi-core machines and in many cases, they contain multiple accelerators, such as GPUs. To fully take advantage of the underlying hardware, DBMS need to exploit the available parallelism, both within and beyond device boundaries. In the past, two conventional approaches have enabled the parallel execution of analytical query plans in multi-CPU environments, Exchange [11] and morsel-driven parallelism [16]. The former encapsulates parallelism and parallelizes standard single-threaded operator implementations, whereas the latter exposes the operators to parallelism and the corresponding shared data structures. However, neither of them supports the parallel execution of queries over heterogeneous hardware because of the lack of system-wide cache-coherence. Pirk et al. [23] propose a declarative algebra that can be compiled to efficient parallel code in either CPU or GPU but does not provide support for concurrent CPU-GPU execution. Chrysogelos et al. [9]

propose HetExchange, a framework that encapsulates the heterogeneous hardware parallelism and allows parallel execution across both CPUs and GPUs concurrently. We adopt the design of HetExchange and extend it with data transfer optimizations to build the analytical engine of our H^2TAP system.

Late materialization. Columnar DBMS are the mainstream option for analytical workloads because they allow queries to access only the required columns. Columnar engines reconstruct intermediate tuples at a later point during option. Tuple reconstruction is either eager, with early materialization, or lazy, with late materialization [1]. Despite the performance penalty for reaccessing columns, late materialization reduces memory accesses for selective queries by deferring the reconstruction of tuples later in the plan. Also, it enables several optimizations such as operations on compressed data and invisible joins [2]. For the case of GPU query processing, data transfers from the main memory, especially over the PCIe bus, incur a high overhead. To alleviate the bottleneck for highly selective queries and to reduce the data movement, Yuan et al. [25, 4] studied the effects of different transfer optimizations on query performance such as compression, invisible joins and transfer overlapping [25]. In this work, we provide a hybrid materialization strategy, show how the different access methods can be used to reduce the volume of transferred data and examine the effect on performance for different interconnect technologies.

Scan Sharing. Traditional DBMS are disk-based and the cost of query processing on them heavily depends on I/O. To improve overall performance, several commercial systems, such as Microsoft SQL Server, RedBrick and Teradata, support optimizations that share disk scans across different queries, consequently reducing the number of I/O requests and amortizing the respective cost. In [12], Harizopoulos et al. define the technique as scan sharing. Furthermore, in [26], Zukowski et al. propose the Cooperative Scans framework that optimizes the scheduling of I/O requests for concurrent scans to maximize bandwidth sharing without latency penalties. We observe that data transfers from the main memory to the GPU are the counterpart of I/O accesses and we adapt the idea of shared scans in this context.

HTAP. Hybrid Transactional and Analytical Processing (HTAP) systems focus on efficiently combining transactional and analytical workloads to provide timeliness in terms of both response time and data freshness. Commercial systems such as DB2, Hekaton, SAP HANA and Oracle DB fall to this category. The HTAP design space forms a spectrum, with the two extremes targetting freshness and isolation, respectively. Freshness-oriented systems, such as the original HTAP of Hyper [14], co-locate the engines and use an efficient snapshotting mechanism to mitigate interference. For the case of Hyper, the snapshotting mechanism is a hardware-assisted copy-on-write (COW). Caldera [3] uses COW to perform HTAP over heterogeneous hardware. Isolation-oriented systems such as BatchDB[17] run the transactional and the analytical engines in different nodes of the system to minimize interference. In BatchDB, the analytical engine applies a delta log, produced by the transactional engine, to update its storage. The frequency of updates defines an epoch, during which a batch of queries shares the same version of the storage. Therefore, the system design sacrifices freshness in favor of isolation.

9. CONCLUSION

In this paper, we evaluate lazy data loading and data transfer sharing techniques in the view of hardware and workload evolution. We describe each one of the approaches, with data transfer sharing being the first time to be used for GPUs in the literature, to the best of our knowledge. Moreover, we provide a non-intrusive HTAP system design which does not restrict the analytical engine to eager or lazy data loading, through a two-tiered, multi-versioned storage. The hardware configuration that we use represents three steps in the evolution of GPU hardware: we start from a slow GPU over a slow interconnect, then we move to a faster GPU over the same interconnect and we end up with the same faster GPU over a faster interconnect. Our analysis reveals that the interconnect remains a bottleneck in several OLAP workloads, and that we can increase its effective bandwidth by applying lazy data loading and data transfer sharing. Finally, we suggest some rules of thumb for tuning GPU-accelerated data management systems, based on the workload.

10. ACKNOWLEDGEMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041) and SNSF project "Efficient Real-time Analytics on General-Purpose GPUs", subside no 200021_178894/1.

References

[1] D. J. Abadi et al. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475, Apr. 2007.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, Vancouver, Canada, 2008.

[3] R. Appuswamy et al. The case for heterogeneous htap. In *CIDR*, 2017.

[4] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, pages 1891–1906, 2016.

[5] S. Breß, H. Funke, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*, 2018.

[6] S. Breß et al. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *CoRR*, abs/1709.00700, 2017. arXiv: 1709.00700. URL: http://arxiv.org/abs/1709.00700.

[7] T. Cao et al. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 265–276, 2011. DOI: 10.1145/1989323.1989352. URL: https://doi.org/10.1145/1989323.1989352.

[8] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *CIDR*, 2019.

[9] P. Chrysogelos et al. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.*, 12(5):544–556, Jan. 2019.

[10] R. L. Cole et al. The mixed workload ch-benchmark. In G. Graefe and K. Salem, editors, *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, page 8. ACM, 2011. DOI: 10.1145/1988842.1988850. URL: https://doi.org/10.1145/1988842.1988850.

[11] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, Baltimore, Maryland, 2005.

[13] B. He et al. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):21:1–21:39, 2009.

[14] A. Kemper and T. Neumann. Hyper: a hybrid oltp olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, Apr. 2011.

[15] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[16] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[17] D. Makreshanski et al. Batchdb: efficient isolated execution of hybrid oltp+olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 37–50, Chicago, Illinois, USA. ACM, 2017.

[18] MapD. https://www.mapd.com/.

[19] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011,*

Athens, Greece, June 13, 2011, pages 17–26, 2011. DOI: 10.1145/1995441.1995444. URL: https://doi.org/10.1145/1995441.1995444.

[20] NVIDIA. License for customer use of nvidia geforce software. https://www.nvidia.com/content/DriverDownload-March2009/licence.php?lang=us&type=GeForce.

[21] P. E. O'Neil et al. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[22] J. Paul, J. He, and B. He. GPL: A GPU-based Pipelined Query Processing Engine. In *SIGMOD*, pages 1935–1950, 2016.

[23] H. Pirk et al. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

[24] P. Sioulas et al. Hardware-conscious Joins on GPUs. In *ICDE*, 2019.

[25] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB*, 6(10):817–828, 2013.

[26] M. Zukowski et al. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 723–734, Vienna, Austria. VLDB Endowment, 2007.