HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics

Hamish Nicholson
hamish.nicholson@epfl.ch
EPFL
Switzerland

Aunn Raza
aunn.raza@epfl.ch
EPFL
Switzerland

Periklis Chrysogelos*
periklis.chrysogelos@oracle.com
Oracle
Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL
Switzerland

ABSTRACT

Accessing input data is a critical operation in data analytics: i) slow data access significantly degrades performance, and ii) storing everything in the fastest medium, i.e., memory, incurs high operational and hardware costs. Further, while GPUs offer increased analytical performance, equipping them with correspondingly fast memory requires even more expensive memory technologies than DRAM; making memory resources even more precious. Existing GPU-accelerated engines rely on CPU memory for bigger working sets, albeit at the expense of slower execution. Such a combination of both memory and compute disaggregation, however, invalidates the assumption of existing caching mechanisms: i) the processing tier is highly heterogeneous, ii) data access bandwidth depends on the access method and compute unit, iii) with NVMe arrays, persistent storage can approach in-memory bandwidth, and iv) all these relative quantities depend on the current query and data placement. Thus, existing caching approaches waste interconnect bandwidth, cache inefficiently, and overall result in suboptimal execution times.

This work proposes HetCache, a storage engine for analytical workloads that optimizes the data access paths and tunes data placement by co-optimizing for the combinations of different memories, compute devices, and queries. Specifically, we present how the increasingly complex storage hierarchy impacts analytical query processing in GPU-NVMe-accelerated servers. HetCache accelerates analytics on CPU-GPU servers for larger-than-memory datasets through proportional and access-path-aware data placement. Our prototype implementation of HetCache demonstrates a 1.14x-1.78x speedup of GPU-only execution on NVMe resident data and achieves near in-system-memory performance for hybrid CPU-GPU execution, while substantially improving memory efficiency. Overall, HetCache turns the multi-memory-node nature of such heterogeneous servers from a burden into a performance booster.

*Work done while the author was at EPFL

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

1 INTRODUCTION

Modern hardware has revolutionalized analytical query processing: NVMe arrays offer persistent storage bandwidth that is comparable to DRAM bandwidth, GPUs offer both high-bandwidth local memory for fast joins, as well as significant processing power [10, 35, 40, 14, 29], and modern CPUs provide both increased performance, as well as a hosting platform for coordinating and connecting the different devices [15]. Yet, as fertile as this new server hardware landscape is, it also invalidates fundamental concepts about data access for analytics.

The problem: access heterogeneity in the processing layer. Existing data caching approaches rely on one or more assumptions that are invalidated in modern NVMe- and GPU-equipped servers. First, many approaches assume that in-memory caching of a frequently-accessed data page is important, while the type of queries accessing it is less important [17, 37, 5, 33, 24, 4, 21]. However, with NVMe arrays providing data access bandwidths that can sustain the processing throughput of many queries, the importance of the query itself increases: caching two pages that have the same access frequency can provide significantly different acceleration depending on whether the query is scan-bound or processing-bound. Second, many approaches assumed that CPUs were the only processors and thus could treat the execution layer as independent of the caching layer. The proliferation of GPUs as analytical coprocessors, however, exacerbates NUMA effects and execution imbalances: faster processors consume higher portions of the input data due to load balancing [6]. As a result, the processing layer is heterogeneous, and haphazardly caching data across the processors may incur significant inter-device communication to achieve balanced execution. Finally, many approaches assumed that only whether a page was cached or not was important for a query. However, the query selectivity, the query access pattern, the relative device throughput, and even the data placement and the interconnects all affect how efficiently each device will access the data pages. As a result, existing approaches result in wasted hardware resources and increased memory capacity requirements.

The solution: execution-centric data caching. Our insight is that most of the wasteful hardware utilization results from caching decisions that have little impact on query execution, while the tension between caching decisions and execution originates from the absence of an appropriate feedback loop across the two layers. To avoid wasteful hardware utilization, we re-evaluate how caching

a page in CPU/GPU memory affects query execution in the presence of both GPUs and NVMe arrays and propose HetCache, a storage engine design that exploits these observations to provide execution-centric data caching on GPU-NVMe servers.

HetCache: Caching & memory efficiency. We see that there are two main contributing factors regarding the caching efficiency: the query type and the relative performance of the consuming devices. Further, there is an input proportion for each query after which caching more input results in diminishing returns [26], and a hardware-query-dependent access granularity below which fine-grained accesses for selective queries boost the caching effect.

HetCache: Caching & accelerators. We pinpoint the tension between caching and execution to the staged decision-making process: the storage layer traditionally places the data into a memory of its choice, which is incompatible with the flexible query execution required for hybrid CPU-GPU execution. HetCache alleviates the tension by enabling a tight integration between the caching layer and the execution's data transfers. Traditionally, the caching layer would decide where to place a page read from storage. This causes the execution layer to further transfer it during execution and requires the caching layer to predict the relative device performance to provide a good split of the data across NUMA nodes, prior to execution. Instead, HetCache lets the execution layer do the data placement and moves the caching layer into a suggestive role. Specifically, the execution does the transfers the more performant way for the current query and the caching layer provides hints to steer the execution layer towards better long-term configurations.

In the rest of this paper, we show how modern hardware (NVMe and GPU acceleration) affects analytical engines and provide HetCache, a blueprint for storage engines that i) efficiently exploit the available memory and storage resources and ii) enable GPU-accelerated analytical engines to benefit from out-of-memory storage and NVMe arrays.

The **contributions** of this paper are:

- Analyzes when and why processing out-of-memory data is a viable alternative, performance-wise, for analytics – and why processing in-memory data is still a requirement for performance (Sections 2 and 3).
- Shows that the choice of storage media depends on granularity and query benefit and that the highly NUMA nature of CPU-GPU-NVMe servers complicates the landscape but provides significant query benefits (Section 4).
- Proposes HetCache, a storage engine design for analytical workloads that envisions an impact-oriented caching mechanism and thereby enables efficient query processing and memory utilization in the presence of heterogeneous CPU-GPU hardware and high-bandwidth storage through proportional and access-path-aware data placement (Section 5).
- We build a prototype implementation of HetCache into Proteus (Section 6) and show that HetCache achieves up to 1.78x speedup for GPU-only execution with NVMe-resident data compared to naive NVMe-GPU transfers and is highly memory-efficient for hybrid CPU-GPU execution (Section 7).

Overall, HetCache turns the multi-memory-node nature of heterogeneous CPU-GPU-NVMe servers from a burden into a performance booster for analytics.

2 SHORTCOMINGS OF CACHING POLICIES

Frequency-based caching. Existing caching policies aim to minimize the number of disk accesses by maximizing the cache hit rate. The most common caching heuristics such as LRU, MRU, and 2Q cache pages based on the frequency and/or recency of use [17, 37, 5, 33, 24]. Such policies assume that processing throughput and input access bandwidth are highly correlated and thus work well when storage is a significant bottleneck. Historically this has been the case due to the significant bandwidth difference between CPU memory and disk bandwidth: even if a query accessed a small number of data pages from storage, the performance penalty was significant. In contrast, recent improvements in interconnect and storage technology significantly reduce the bandwidth gap: aggregating NVMe drives into arrays can offer storage bandwidth comparable to DRAM bandwidth. As a result, the benefit of caching a page depends on its use: caching a page participating in a query that is slow due to another operation will provide a small benefit compared to caching a page for an input-IO bound query. Even if the pages are accessed with the same frequency. **Caching policies that treat pages consumed by slow and fast queries as equal wastes memory capacity on servers with high bandwidth storage.**

The more-caching-the-better. Most past approaches treat all pages of a (disk-resident objects[1], access frequency)-combination as equal and will try to cache as many pages as possible. However, as disk bandwidth increases relative to query processing throughput, the following scenario occurs: even for a column where caching is beneficial, as more pages are cached, we cross a threshold where input-IO bandwidth ceases to be the bottleneck and caching more pages provides diminishing performance improvements. As a result, HPCache [26] caches column proportions; for CPU-NVMe execution, HPCache finds a balance between capacity, optimal column caching proportion, and expected query patterns. Still, HPCache assumes a linear storage hierarchy and does not handle either multiple transfer paths or heterogeneous compute devices, which have varying query processing performance, **Caching benefits are no longer linear to the size of the cache; the point of diminishing returns for caching depends on both how much and where data is cached.**

Centralized caching for homogeneous architectures. The existing centralized caching process assumes a uniform and centralized, shared-everything architecture inside the server, with a single processing unit type, the CPU. Yet, modern servers have multiple CPU sockets and processing unit types (e.g., GPUs). As a result, such architectures have multiple access paths, each with different bandwidths. Further, the multiple processing units invalidate the single processing throughput assumption: GPUs process queries at different (slower or faster, depending on the case) rates than the CPU. Exacerbating this, hybrid CPU-GPU execution may also distribute data across the different devices for load-balancing purposes, making the overall system throughput dynamic. Lastly, memory is heterogeneous and distributed across multiple devices: GPU memory has a different access profile than CPU memory, and both are distributed across multiple chip(let)s. As a result, caching data in one memory node provides a potentially different benefit from

[1] Columns or segments, depending on the specific system.

caching it in another node, depending on the interconnect congestion, the relative processing unit, query processing throughput, and even the access granularity. **Heterogeneity breaks uniformity assumptions about the processing and caching layer of adaptive caching mechanisms.**

The simplifying alternative: no caching for analytics. Given the multitude of challenges related to caching and the high-bandwidth storage alternatives, past work has also suggested just excluding analytical queries from caching [3]. These caching policies, while potentially simplifying analytical query processing, rely on the assumption that analytics are either unpredictable or very slow already. Still, with warehouses and more data-demanding applications, analytical queries have become more popular. Further, with the introduction of NUMA-aware, parallel and GPU-accelerated solutions, analytics can reach response times of a few milliseconds for hundred of GBs of inputs. As a result, some queries see significant performance degradation due to waiting on out-of-memory data fetches. **High-performance analytics requires, at least some, memory for caching.**

3 A PITSTOP BEFORE THE DISK

Despite the availability of high bandwidth storage, *not all data should be stored on block storage devices.* In this section, we argue when and why it is still necessary to store input data in memory. In the scope of this paper, we do not consider spilling intermediate data, such as hash tables, to block storage.

3.1 CPU is not always slow

CPU workloads can still be bottlenecked by scans, even in memory. Figure 2 shows the consumption throughput of a query that scans a single column, applies a selective filter and performs a summation. In order to apply the predicate, the query must read each value in the column. The black line shows the query consumption throughput when the data is on NVMe storage as the available storage increases. The dashed blue line shows the consumption throughput of the same query when the column is fully memory resident. The query processing throughput of NVMe resident data scales linearly with the available storage bandwidth until the storage bandwidth exceeds half of the memory bandwidth (measured to be 126 GB/s). This is because in addition to the CPU reading the data from memory, each NVMe transfer also consumes memory bandwidth in order to write (stage) the data to memory[2]. In our test bed, the filter reads from memory and not the CPU's last level cache; thus, as the storage bandwidth approaches half of the memory bandwidth, the query becomes memory-bandwidth bound. In contrast, when the data is fully memory resident, the query is the only thing reading or writing to memory and can read the data at full memory bandwidth. **Query execution on NVMe-resident data can be bottlenecked by memory bandwidth.**

Example. Arrays of NVMe storage have a bandwidth greater than the processing throughput of many queries when executed on CPUs [26]. Figure 1a plots the execution time of two Star Schema Benchmark (SSB) [27] queries when the fact table is on disk and the storage bandwidth increases from 7 GB/s to 86 GB/s. [3] Both of these queries construct in-memory hashtables and then access the fact table sequentially to probe the join hashtable. The execution time of the two queries on memory-resident data is shown in dashed lines. Query 1.3 consumes input data at a relatively high input bandwidth and executes faster as the available storage bandwidth increases. In contrast, query 3.1 consumes data at a lower rate as the query processing is more complex, involving multiple joins and grouping. It only improves up to 14 GB/s because, beyond this, storage is not the bottleneck in query execution. High bandwidth storage shifts the bottleneck away from storage for coarse-grained accesses.

3.2 GPU acceleration

GPUs significantly accelerate analytical query processing. When data is in local memory, GPUs execute queries up to 25x faster than CPUs [35, 40, 14, 10]. This is due to both the higher bandwidth memory available on GPUs, up to 2 TB/s on a current-generation Nvidia A100 GPU, and the threading model, which enables GPU execution to mitigate the cost of memory stalls. However, such performance requires the data to be GPU resident prior to processing. While GPU execution may still be beneficial for processing non-GPU-resident data, it is predominantly bottlenecked by the data transfer over the interconnect [32, 18, 7, 40]. Caching can alleviate this bottleneck, but cache sizes are constrained by the comparatively small GPU memory capacity. Hence, GPU memory must be used judiciously to cache data with the largest impact on query execution times. **GPU acceleration needs memory efficient caching to mitigate data transfer bottlenecks.**

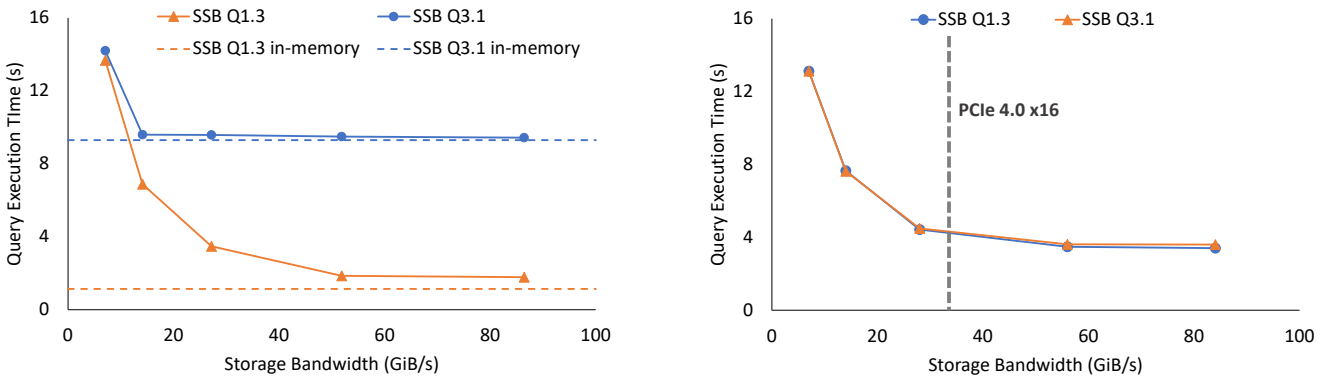
Example. However, bandwidth to input data is still a bottleneck for GPU-accelerated query execution. Arrays of NVMe drives can have a bandwidth that far exceeds the relatively limited interconnect bandwidth of a GPU. Figure 1b shows the execution time for the same setup but when executing on a GPU. Both query 1.3 and query 3.1 improve as the storage bandwidth increases until 32 GB/s, which is the maximum bandwidth of the PCIe 4.0 x16 connection that the GPU is equipped with. The vertical dashed orange line shows the PCIe 4.0 x16 bandwidth. Even with this bottleneck, queries such as 3.1 improve over CPU execution due to the GPU's high memory bandwidth and the high number of concurrent contexts. However, queries such as 1.3 perform worse than on the CPU as the CPU can scan the data faster since it can read data from the NVMe drives at a greater bandwidth. Naive NVMe-GPU transfers do not fully exploit the bandwidth of NVMe arrays.

4 HETEROGENEOUS MEMORY HIERARCHY

In-memory data caching provides significant acceleration for some analytical workloads; however, it is also expensive as memory capacity comes at a significant cost [39]. In contrast, out-of-memory storage 1) is significantly cheaper with a better price/performance ratio [22], 2) continues to get cheaper relative to DRAM [13], but 3) is also slower. However, fast and slow need to be considered relative to the workload requirements. While storing data in memory can improve query response times, not all workloads benefit equally

[2]This is micro-architecture dependent. Intel Xeons feature DDIO, which transparently enables IO devices to read/write directly from the CPU last-level caches bypassing DRAM, in some cases reducing the memory bandwidth requirements of IO [1]. To the best of our knowledge, AMD CPUs do not have an equivalent feature. Our testbed (Section 7) uses an AMD EPYC CPU.

[3]The storage bandwidth is controlled by varying the number of NVMe drives the data is striped across.



(a) CPU-only execution

(b) GPU-only execution

Figure 1: Execution time as the available storage bandwidth increases for two SSB queries at scale factor 1000. See Section 7 for the full experimental setup.

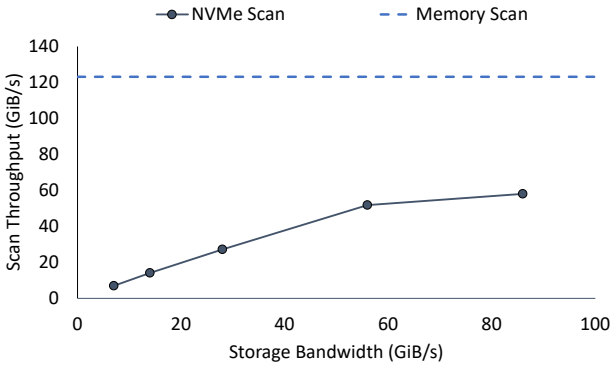


Figure 2: Scan throughput for scan-filter-aggregate query with varying storage bandwidth. See Section 7 for the full experimental setup.

from in-memory storage. Thus, treating all the input data equally for in-memory caching may waste memory for little-to-no benefit. Conversely, out-of-memory processing can significantly slow down some workloads. Further, with the proliferation of accelerators, the performance benefit of storing data in memory depends on both the compute device consuming it as well as the difference in bandwidth between memory and out-of-memory storage. Efficient use of memory demands a workload-aware approach.

Traditionally, storage systems follow a linear hierarchy. Data is loaded into faster mediums before being processed; the CPU loads data from NVMe to DRAM, and then the CPU loads the data from memory to its caches and, ultimately, registers for processing. However, heterogeneous servers do not maintain a linear hierarchy. Data can be copied between CPU and GPU memory at cache-line or greater granularity, and GPUs can also directly access CPU memory at fine granularity [8]. Both CPUs and GPUs can directly transfer data from NVMe drives to their local memory. Further, data can be transferred from point to point via multiple intermediate transfers. For example, data can flow from NVMe storage to GPU memory through CPU memory.

The diversity in available access paths and the non-uniform query processing throughputs challenges the caching design of DBMS. Caching policies must determine both when and *where* to cache. Individually, each compute device achieves maximum query processing throughput when consuming data from local memory. For hybrid CPU-GPU execution, the ideal data placement across devices is proportional to their query processing throughput. However, there is a *capacity* constraint, which is non-uniform across devices; CPU memory is much larger than GPU memory, yet GPUs typically have greater query processing throughput. Thus, once the GPU cache is full, GPU query execution depends on efficient data transfers. This data can be transferred from CPU memory or NVMe storage, and both can saturate the interconnect. The key difference between the two options is the *access granularity*. GPUs can directly access CPU memory at fine granularity but can only access NVMe storage at the block level. Because many queries selectively access columns, e.g., due to joins or filters, CPU-memory caching for GPU consumption can enable faster query response times due to the reduction in IO-amplification compared to transferring full pages.

Figure 3 demonstrates the impact on the performance of pushing whole pages to the GPU compared to accessing values directly in CPU memory as the selectivity varies. For this micro-benchmark, the input data is a table of two columns of uniformly distributed integers. Each column is 100 GB. The query filters tuples using the first column and then sums the values from the second column for tuples that pass the filter using late materialization. The filter is used to control the selectivity. For GPU execution, directly accessing values in CPU memory is advantageous for selectivities below 10% compared to transferring all the data using 2MiB pages. However, for higher selectivities, it is more performant to push the data via 2MiB pages.

CPU memory can serve as a cache for pages selectively accessed by GPU execution: the selectivity reduces the amount of data transferred over the interconnect during late materialization for CPU-resident data. Using CPU memory to cache data for the GPU is viable because, for many queries, CPU execution is slower than storage bandwidth and therefore does not require a large cache to maximize the CPU's own processing throughput. However, CPU memory to GPU transfers also consumes CPU memory bandwidth. This may cause interference in CPU execution for bandwidth-intensive

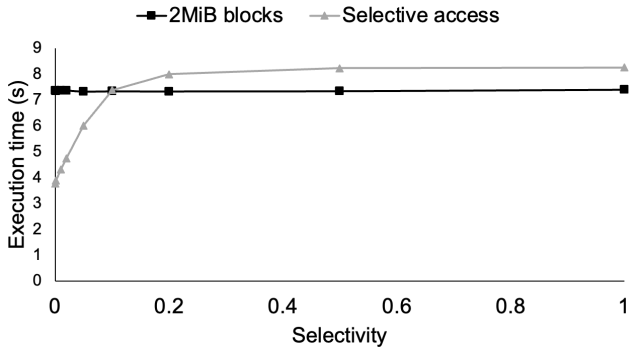


Figure 3: Execution time for a scan-filter-aggregate query as the selectivity of the filter is varied. See Section 7 for the full experimental setup.

queries. Caching in a heterogenous memory hierarchy is a trade-off between access granularity, capacity, and bandwidths. Effective caching policies must consider all three properties to meet the bandwidth demands of the workload.

5 HETCACHE: WORKLOAD AND HARDWARE AWARE STORAGE

We envision HetCache, a storage engine that efficiently utilizes the memory and storage resources of NVMe-CPU-GPU servers to accelerate query execution on larger-than-memory data. HetCache is workload-aware in order to determine when to cache data, utilizing memory to only cache columns that will accelerate query response times. Further, HetCache holistically considers the query throughput of CPUs and GPUs as well as the per-query per-column access selectivity to determine where to cache data and which transfer path to use.

5.1 Staged SemiLazy Transfers

GPU query throughput for non-GPU resident data is primarily bottlenecked by the interconnect. Both memory and storage bandwidths exceed the GPU-system interconnect bandwidth, and in most cases, the processing throughput of GPU query execution also exceeds the interconnect bandwidth. Input data can either be eagerly (pushed) or lazily transferred (pulled) to GPUs. Eager transfers push full pages of data over the interconnect into GPU memory before the GPU begins processing those pages, overlapping processing and data transfers as through HetExchange [6]. Eager transfers result in a sequential access pattern, enabling full utilization of the interconnect bandwidth. GPU query execution on GPU resident data benefits from both the high bandwidth and low latency of GPU memory data accesses. However, not all values pushed across the interconnect may be accessed by the query. For example, values from columns that are accessed after a series of selective conditions, such as joins and filters, have a lower probability of being accessed by the query. Modern GPUs can use a unified virtual memory address space with CPU memory, enabling GPUs to directly access CPU memory at a fine granularity but at higher latency than accessing GPU-local memory [8]. Lazy transfers are beneficial when few values are required per page, as less data needs to be transferred across the interconnect. SemiLazy transfers [32] eagerly transfer pages for columns that need to be nearly entirely accessed and lazily transfer values from pages that are selectively accessed.

HetCache mitigates the interconnect bottleneck through *Staged SemiLazy Transfers*. Lazy transfers require the accessed data to be in CPU memory. Based on per-column query selectivity hints, HetCache handles GPU page requests for NVMe-resident data by either eagerly moving the page from NVMe storage to GPU memory or staging the page in CPU memory through an NVMe to CPU memory transfer. The GPU can then lazily fetch values in the page at a finer granularity resulting in more efficient utilization of the GPU interconnect. Staged SemiLazy transfers result in better storage bandwidth utilization, as the full storage bandwidth is available for NVMe to CPU memory transfers, while NVMe to GPU transfers is limited by the GPU interconnect.

5.2 Heterogeneity-aware Caching

HetCache implements a heterogeneity-aware caching policy. It uses both workload information, query processing throughput, and selectivity hints, as well as hardware information, interconnect storage, and memory bandwidths, to determine how much and where to cache.

To determine how much to cache on each device for device-local access, HetCache observes the per-device query processing throughput. The rate of page requests that a query makes from each device is used to infer the query's per-device processing throughput. With the per-device processing throughput, the known interconnect, storage, and memory bandwidths, HetCache determines the maximum amount of each column to be cached on each device without caching any column beyond the point of diminishing returns. i.e., it aims to match the effective bandwidth to query inputs, which can be partially in memory and partially in storage, to the query processing throughput.

HetCache also uses CPU memory to cache data for GPU accesses. As CPU memory capacity is large compared to query processing throughput, CPUs typically experience less memory pressure than GPUs which have higher query processing throughputs and constrained memory capacities. Staged SemiLazy caching can move the bottleneck from the interconnect back to storage for queries that selectively access columns. Hence, for these queries, it is necessary to cache pages from selectively accessed columns in the higher bandwidth CPU memory to utilize the GPU interconnect fully. HetCache uses both the per-device query throughput and the selectivity hints to determine if GPU query processing is interconnect-bandwidth-bound or storage bandwidth-bound. In both cases, it will cache pages in GPU memory until limited by GPU memory capacity. Once constrained by the GPU memory capacity, HetCache will preferentially cache densely accessed pages in GPU memory and cache sparsely accessed pages for storage-bandwidth bound queries in CPU memory, optimizing for overall GPU query processing throughput whether queries are bottlenecked by the interconnect bandwidth or by storage bandwidth.

6 SYSTEM

We use Proteus as our analytical engine [6, 32]. Proteus is an in-memory execution engine with support for parallel query execution

across mixes of CPUs and GPUs. Query pipelines are parallelized by instantiating a pipeline instance per compute unit (CPU core or GPU). A logical scan operator emits page handles that are then routed to query pipeline instances. The routing policies are pluggable, defaulting to a prefer-local routing policy. The prefer-local routing policy routes page handles (in rowsets) to pipeline instances running on a compute unit local to the memory referenced by the corresponding rowset, with a fallback option of using a non-local compute unit if the local one is busy. When data transfers are eager, before touching the tuples inside a page, a mem-move operator [6] transfers the page to the consumption site if it's not already there. Memory transfers are scheduled asynchronously to compute tasks to overlap data transfers and intra-device execution.

Integrating HetCache into Proteus required handling the late binding of input pages to memory. Specifically, before HetCache, Proteus, as an in-memory engine, expected the data to already be populated in memory and a page handle to refer to a single actual data page – with only the exception of snapshot-related copies [32]. Any copies of the input data, e.g., due to a transfer from one device to another, were transient and only existed for the required subset of the query lifetime, i.e. until they were consumed and the memory reclaimed for another data transfer. Further, when the mem-move operator received a page handle, it would check its current placement. If the current location was acceptable (i.e., for an eager transfer if it was already in the destination node, or for a lazy transfer if it was accessible by the target node), and if not, it would move it to the appropriate node.

In contrast, HetCache, similar to traditional storage engines, can result in two or more copies of a data page: one in storage (NVMe) and potentially one or more copies in the block manager (in-memory cached page, potentially replicated across CPU/GPU memory). Having multiple (immutable) copies of the same page provides additional freedom for memory transfers but also challenges. We implement a routing policy that consults HetCache on the locations of the pages in a rowset. Rowsets are routed based on the locations of the first column in the rowset since it is the most likely to be cached in GPU memory, as Proteus will access all the values from the first column. Recall that caching for CPU requests does not consider the selectivity; thus, there is no bias towards caching the first column in CPU memory. If the page is in both CPU and GPU memory, the rowset is routed to a GPU pipeline instance to leverage the GPU's greater processing speed on cached data. We updated the mem-move to consult with HetCache on three things: 1) the location of copies of the corresponding page, 2) whether the transferred page would be cached in its target node for subsequent queries, and 3) if yes, whether the transferred page should be cached for selective access or not. HetCache's answers to (2) and (3) are treated as hints, and the mem-move is allowed to ignore them. Based on the three answers, the mem-move decides whether a transfer is necessary or if an existing in-memory copy of the page is suitable. If a transfer is necessary, it selects a target memory node, schedules the transfer, and on its completion, it registers the new page location with HetCache along with the ID of the query making the request and a selectivity hint passed down from the query plan. The query ID is used by HetCache to infer the query's processing throughput. Then, HetCache can decide whether it wants to keep the resulting page as part of the cache. To maintain the page in the cache, we use reference counting: each data page is accompanied by a reference count for the flow-based execution; we extend that and keep one extra reference in HetCache to prevent the page from being evicted/released when it is consumed. To evict a page, HetCache releases its reference, though this may not immediately free the page's memory as query pipelines may still hold temporary references. At the moment that the mem-move decides to do a data transfer, it has derived a source and a target location; still, there are a set of different actions required to implement the transfers. Specifically, the mem-move inspects the source and target location and invokes the block or the storage manager to do the actual data transfers, depending on whether it's a memory-only or NVMe-involved data transfer.

7 EVALUATION

This section includes the results of our experimental evaluation. First, we describe the hardware, some specifics of the transfers, and the benchmark used in our experiments. Then we present the results for GPU execution for varying access paths and initial data locations, demonstrating the performance benefit of indirect NVMe-GPU transfers. Finally, we evaluate the data placement of HetCache for hybrid CPU-GPU execution and show the memory-efficiency benefits of workload and hardware aware caching.

7.1 Experimental setup

Hardware. All experiments were conducted on a server with a 2x24-core AMD EPYC 7413 processor, having two threads per core, totalling 96 threads and 256 GB of DRAM. Each CPU socket is connected with a single Nvidia A40 GPU with 48 GB memory using 16 PCIe 4.0 lanes and 12 Corsair MP600 Pro NVMe drives, each using 4 PCIe 4.0 lanes. We observe a maximum memory bandwidth of 116 GB/s per socket on the STREAM triad benchmark [23], and 86 GB/s sequential read bandwidth from all 12 NVMe when using fio [2]. All experiments were conducted on a single socket and utilized all 12 NVMe drives unless otherwise stated.

Software. CPU-GPU memory page transfers use CUDA's memcopy mechanism, while the NVMe-involved transfers branch based on whether the target is CPU or GPU memory: For NVMe to CPU-memory IO, we use io_uring [9], and for direct NVMe to GPU IO, we use Nvidia's GPU-Direct storage API [11]. All page-level transfers use 2 MiB as the IO transfer size to align with the 2 MiB huge pages used by Proteus. Lazy sub-page data accesses happen as DMA requests, and we enforce the accessed page to be present in CPU memory to be globally accessible.

Benchmark. We evaluate our methods using the Star Schema Benchmark (SSB) [27] with a scale factor (SF) of 1000. SSB has four query groups, and within each query group, the query's selectivity decreases with the rank; for example, Q1.3 is more selective than Q1.2, which is more selective than Q1.1. We store the data in binary columnar format, resulting in Query groups 1-3 having a working set of 96 GB per query and group 4 having a working set of 144 GB.

7.2 Staged SemiLazy transfers

Figure 4 analyzes the performance of SemiLazy transfers from memory and staged SemiLazy transfers from NVMe compared to eager transfers under GPU-only execution. The experiment executes SSB

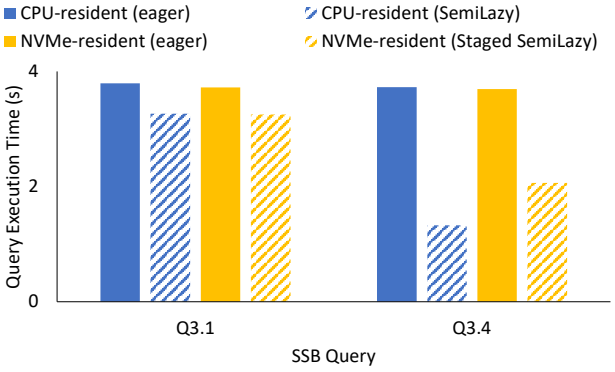


Figure 4: Comparison of transfer paths for GPU-only execution with CPU-memory resident data (blue) and NVMe resident data (yellow)

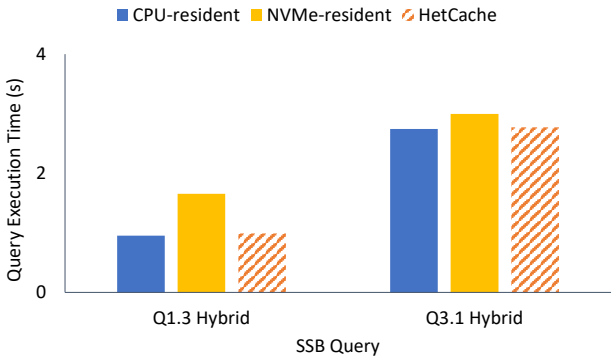


Figure 5: Query processing on CPU-GPU with CPU-resident, NVMe-resident, and HetCache'd data

Q3.1 and Q3.4 at SF1000; the queries have selectivities of 3.4% and 0.000076%, respectively. In both cases, whether the data is read from CPU memory or NVMe directly, eager transfers are bottlenecked by the GPU interconnect for both queries; this is because, regardless of query selectivity, the entire query's working set (96 GB) is transferred to GPU, although the transfers are overlapped with processing. Although SemiLazy transfers less data over the interconnect, it does so at the cost of higher memory latency from CPU memory. The benefit of SemiLazy transfers is more apparent in Q3.4 compared to Q3.1 because Q3.4 is more selective, and thus, less data is transferred over the interconnect for the selectively accessed columns. However, in the case of Staged SemiLazy transfers, the lower selectivity causes the query bottleneck to shift from the interconnect to the NVMe transfers. Although less data is transferred across the interconnect, the selectively accessed columns must still be loaded from the NVMe drives into CPU-memory.

In summary, SemiLazy and staged SemiLazy transfers enable GPUs to access data at a finer granularity (directly accessing values in CPU memory compared to disk blocks) and thereby reduce data transferred over the interconnect in the case of selective accesses, and for very selective accesses, shifts the bottleneck from GPU interconnect to storage. Staged SemiLazy transfers enable the DBMS to optimize transfer granularity, capacity, and limited interconnect bandwidth.

7.3 Workload-aware caching

Figure 5 plots the execution time of SSB Q1.3 and Q3.1 at scale factor 1000, executed on both, CPU and GPU (HetExchange [6]), with (staged) SemiLazy transfers. We set the CPU and GPU maximum cache sizes to 90 GB and 10 GB, respectively. We plot the execution times when the data is completely NVMe resident, completely CPU memory resident, and with warm HetCache-managed caches. Q1.3 is bandwidth-intensive and thus benefits from data being in memory. In comparison, Q3.1 is more compute-intensive and only suffers a 5% penalty when accessing data directly from NVMe. Q1.3 requires 75 GB of CPU cache, in addition to 10 GB of GPU cache, to approach fully CPU-memory resident performance, while Q3.1 requires only the full 10 GB of GPU-memory cache and no CPU-memory. This is due to the fact that query pipelines being executed on GPU are bottlenecked by the interconnect bandwidth, while for Q1.3 the CPU query processing performance is the same whether the data is initially NVMe-resident or CPU-memory resident.

Figure 6 analyzes the effect of CPU cache size on Q1.3 by plotting execution time with varying cache sizes in CPU memory while having a fixed 10 GB GPU cache. Recall that Q1.3 is bandwidth-intensive and hence fully uses a 10 GB GPU cache and requires up to 80% of data in CPU memory before there are diminishing returns on caching any more input data.

Impact-oriented caching through proportional caching allows HetCache to tune the amount of cached data to best effort to satisfy the query's consumption bandwidth requirements but not beyond the point of diminishing returns. Q3.1 represents the class of queries that do not require caching data in CPU memory given that the available storage bandwidth is higher than the query's bandwidth requirement for CPU execution, and thus, saves CPU memory for caching other data, which may have a higher impact on the overall performance of the analytical engine. While Q1.3 is representative of class queries that are bandwidth-intensive and benefit more from caching in faster memory; in this case, the query's execution time improves until 80% of input data is cached. Further, workload awareness allows the HetCache to select appropriate transfer methods across devices, overcoming the common bandwidth wall for accelerated query processing. Queries with lower selectivities benefit from staging data in CPU-memory, getting the best of both worlds; high-bandwidth loading from NVMe to CPU-memory while efficiently using CPU-GPU interconnect via granular accesses. In summary, HetCache introduces workload-aware and impact-oriented caching and efficiently utilizes the storage tiers for maximum performance.

8 RELATED WORK

Buffer managers optimized for nearly in-memory processing. Recent work has minimized the overheads of buffer managers when processing in-memory data. This enables in-memory performance when the working set fits in memory but also graceful performance degradation when the working set exceeds memory capacity. Providing persistency and support for out-of-memory data traditionally introduces two overheads with respect to the buffer pool. First, having a centralized buffer pool creates a point of contention [16]. Second, persistence requires a level of indirection when translating in-memory references to out-of-memory object references. Graefe

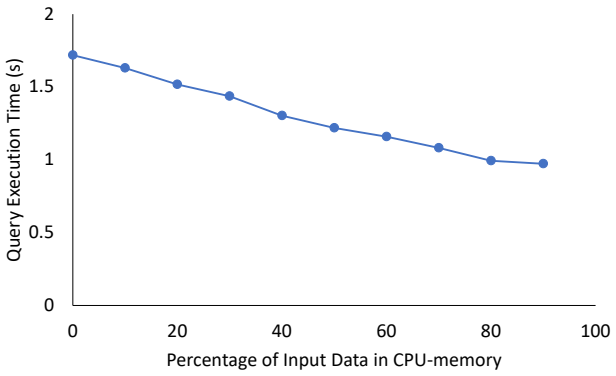


Figure 6: Execution time of SSB Q1.3 at SF1000 with a fixed 10 GB GPU cache and varying size CPU memory cache.

et al. [12] use pointer swizzling to eliminate buffer pool overheads when all data fits in memory. By avoiding a hashtable, they avoid a costly central point of contention. LeanStore [19] extends on pointer swizzling by speculatively unswizzling pages, keeping hot pages in memory without explicitly tracking page accesses in a shared data structure. Umbra [25] extends LeanStore with support for variable-length buffer frames, improving the handling of large objects. Optimizing buffer pool accesses reduces the overhead imposed on mostly-in-memory analytics while adding support for out-of-memory data. Unlike HetCache, these approaches strive to keep the working set in memory, regardless of the bandwidth required by the workload. These methods are complementary to HetCache, which improves the cache efficiency with respect to the performance gains achieved by caching data in memory but does not attempt to improve upon the buffer manager overhead.

GPU data transfers. HetCache is not the first system designed to overcome the GPU interconnect bottleneck. To alleviate the interconnect bottleneck for highly selective queries, Yuan et al. studied the effects of different CPU-GPU transfer optimizations on GPU query performance, including compression, invisible joins, and transfer overlapping [40]. Raza et al. introduce SemiLazy transfers to GPU for system memory resident data [32]. HippogriffDB [20] uses workload-aware adaptive compression to maximise the effective bandwidth of data transfers to GPUs and also introduces direct NVMe-GPU transfers bypassing CPU memory. Data is stored in memory or on disk in a compressed format and is decompressed by the GPU just before it is consumed by query execution. BaM [31] introduces a technique to enable GPU orchestration of IO requests, bypassing the CPU for both orchestration and data transfers. This enables GPUs to access NVMe resident data with disk-block-sized data-dependent access patterns performantly.

Integrating GPUs into the storage hierarchy. Most GPU-accelerated database systems either operate only on GPU-resident data or transfer data from system-memory to GPU-memory at query execution time [29, 34]. GPU memory can either be treated as a peer to CPU memory or as a higher level of the storage hierarchy. Systems such as AresDB [36] and HeavyDB [38, 28] take the latter approach, using CPU memory as a staging area for all transfers to GPU memory. HippogriffDB treats GPU memory as a peer to CPU memory. It streams input data from both CPU memory and NVMe storage to the GPU for query processing [20]. PG-Strom [30] can load data directly from NVMe to a GPU as well as from CPU memory. While it implements a GPU cache, it does so at the table level of granularity, limiting its utility to tables smaller than GPU memory.

9 CONCLUSION

This paper shows how modern hardware (NVMe and GPU acceleration) affects analytical engines and demonstrates that data placement at each layer in the heterogenous memory hierarchy is a trade-off between: 1) *Bandwidth:* Each memory tier can access data at different bandwidths. The maximum bandwidth tier is not necessarily optimal. It can use capacity without performance gain at the expense of other queries, which could benefit from increased bandwidth. 2) *Capacity:* Data stored in device-local memory results in the highest performance. However, with the constrained capacity of each device, not all data can be stored in local memory. Heterogenous processing throughput and capacity means high-throughput capacity-limited devices can cache in other devices' memory. 3) *Access granularity:* Even if two access paths offer the same bandwidth, the access granularity can limit the effective bandwidth due to IO amplification. Indirect transfers from block-level devices via an additional cache-line addressable device can mitigate the cost of IO amplification due to non-uniform access to storage bandwidth.

We propose HetCache, a storage engine design for analytical workloads. HetCache encapsulates heterogeneity through impact-oriented proportional caching and access-path-aware data placement. Our prototype implementation of HetCache achieves up to 1.78x speedup of GPU-only execution on NVMe resident data, and HetCache-managed caches can achieve in-CPU-memory performance with hybrid CPU-GPU execution without storing all data in memory.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback. This work was partially funded by SNSF project "Efficient Real-time Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

REFERENCES

[1] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. 2020. Data Direct I/O Characterization for Future I/O System Exploration. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. (August 2020), 160–169. DOI: 10.1109/ISPASS48437.2020.00031.

[2] Jens Axboe. 2022. Fio. original-date: 2012-10-22T08:20:41Z. (January 2022). Retrieved 01/26/2022 from https://github.com/axboe/fio.

[3] W Bridge, A Joshi, M Keihl, T Lahiri, J Loaiza, and N Macnaughton. 1997. The Oracle Universal Server Buffer Manager. en. In *Proceedings of the 23rd VLDB Conference*. Athens, Greece, 5.

[4] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD bufferpool extensions for database systems. en. *Proceedings of the*

VLDB Endowment, 3, 1-2, (September 2010), 1435–1446. DOI: 10.14778/1920841.1921017.

[5] Hong-Tai Chou and David J. DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1, 3, 311–336. DOI: 10.1007/BF01840450.

[6] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. en. *Proceedings of the VLDB Endowment*, 12, 5, (January 2019), 544–556. DOI: 10.14778/3303753.3303760.

[7] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in gpu-accelerated analytical engines. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[8] 2020. CUDA C++ Programming Guide. en, 379.

[9] 2019. Efficient IO with io_uring. (2019). Retrieved 12/10/2022 from https://kernel.dk/io_uring.pdf.

[10] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. en. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston TX USA, (May 2018), 1603–1618. DOI: 10.1145/3183713.3183734.

[11] 2019. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. en-US. (August 2019). Retrieved 12/10/2022 from https://developer.nvidia.com/blog/gpudirect-storage/.

[12] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-memory performance for big data. en. *Proceedings of the VLDB Endowment*, 8, 1, (September 2014), 37–48. DOI: 10.14778/2735461.2735465.

[13] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[14] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. en. *ACM Transactions on Database Systems*, 34, 4, (December 2009), 1–39. DOI: 10.1145/1620585.1620588.

[15] 2022. HPC Tuning Guide for AMD EPYC 7003 Series Processors. en. Technical report.

[16] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. en. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. ACM Press, Saint Petersburg, Russia, 24. DOI: 10.1145/1516360.1516365.

[17] Theodore Johnson and Dennis E. Shasha. 1994. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors. Morgan Kaufmann, 439–450.

[18] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. en. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware - DaMoN '12*. ACM Press, Scottsdale, Arizona, 55–62. DOI: 10.1145/2236584.2236592.

[19] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. en. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, (April 2018), 185–196. DOI: 10.1109/ICDE.2018.00026.

[20] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. en. *Proceedings of the VLDB Endowment*, 9, 14, (October 2016), 1647–1658. DOI: 10.14778/3007328.3007331.

[21] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, and Lihua Yue. 2009. CCF-LRU: a new buffer replacement algorithm for flash memory. *IEEE Transactions on Consumer Electronics*, 55, 3, (August 2009), 1351–1359. Conference Name: IEEE Transactions on Consumer Electronics. DOI: 10.1109/TCE.2009.5277999.

[22] David Lomet. 2018. Cost/performance in modern data stores: how data caching systems succeed. en. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. ACM, Houston Texas, (June 2018), 1–10. DOI: 10.1145/3211922.3211927.

[23] John D. McCalpin. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, (December 1995), 19–25.

[24] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (FAST '03). USENIX Association, USA, (March 2003), 115–130.

[25] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[26] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. en. In *DaMoN'22*. Association for Computing Machinery, Philadelphia, PA, USA, 9. DOI: 10.1145/3533737.3535100.

[27] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, 237–252.

[28] 2019. OmniSciDB Developer Documentation — OmniSciDB documentation. (2019). Retrieved 09/01/2022 from https://heavyai.github.io/heavydb/.

[29] Johns Paul, Shengliang Lu, and Bingsheng He. 2021. Database systems on GPUs. *Foundations and Trends® in Databases*, 11, 1, 1–108. DOI: 10.1561/1900000076.

[30] 2021. PG-Strom Manual. (2021). Retrieved 04/09/2022 from https://heterodb.github.io/pg-strom/.

[31] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I.-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2022. BaM: A Case for Enabling Fine-grain High Throughput

GPU-Orchestrated Access to Storage. en. arXiv:2203.04910 [cs]. (March 2022). Retrieved 08/31/2022 from http://arxiv.org/abs/2203.04910.

[32] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. Gpu-accelerated data management under the test of time. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[33] Allen Reiter. 1976. A Study of Buffer Management Policies for Data Management Systems. Technical report. WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER.

[34] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. en. *ACM Computing Surveys*, 55, 1, (January 2023), 1–38. DOI: 10.1145/3485126.

[35] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. en. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, (June 2020), 1617–1632. DOI: 10.1145/3318464.3380595.

[36] Jian Shen, Ze Wang, David Wang, Jeremy Shi, and Steven Chen. 2019. Introducing AresDB: Uber's GPU-Powered Open Source, Real-time Analytics Engine. (January 2019). Retrieved 09/01/2022 from https://www.uber.com/blog/aresdb/.

[37] Michael Stonebraker, John Woodfill, Jeff Ranstrom, Marguerite C. Murphy, Marc Meyer, and Eric Allman. 1983. Performance enhancements to a relational database system. *ACM Trans. Database Syst.*, 8, 2, 167–185. DOI: 10.1145/319983.319984.

[38] Mostak Todd. 2013. An Overview of MapD (Massively Parallel Database). Technical report. Massachusetts Institute of Technology.

[39] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. en. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, (February 2022), 609–621. DOI: 10.1145/3503222.3507731.

[40] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. en. *Proceedings of the VLDB Endowment*, 6, 10, (August 2013), 817–828. DOI: 10.14778/2536206.2536210.