Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs

Bobbi Yogatama
University of Wisconsin-Madison
bwyogatama@cs.wisc.edu

Weiwei Gong
Oracle Corporation
weiwei.gong@oracle.com

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

ABSTRACT

GPU-accelerated databases have been gaining popularity in recent years due to their massive parallelism and high memory bandwidth. The limited GPU memory capacity, however, is still a major bottleneck for GPU databases.

Existing approaches have attempted to address this limitation by using (1) hybrid CPU-GPU DBMS or (2) multi-GPU DBMS. We aim to improve prior solutions further by leveraging both hybrid CPU-GPU DBMS and multi-GPU DBMS at the same time. In particular, we explore the design space and optimize the *data placement* and *query execution* in hybrid CPU and multi-GPU DBMS. To improve data placement, we introduce the *cache-aware replication policy* which takes into account the cost of shuffle when replicating data and could coordinate both caching and replication decisions for the best performance. To improve query execution, we extend the existing hybrid CPU-GPU query execution strategy with distributed query processing techniques to support multiple GPUs. We build a system called *Lancelot*, a hybrid CPU and Multi-GPU data analytics engine with all the optimizations integrated.

Our evaluation shows that the *cache-aware replication* outperforms other policies by up to 2.5× and Lancelot outperforms existing GPU DBMSes by at least 2× on Star Schema Benchmark and 12× on TPC-H Benchmark.

PVLDB Reference Format:
Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. PVLDB, 17(13): 4709 - 4722, 2024.
doi:10.14778/3704965.3704977

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitfront.io/r/bwyogatama/X3t8gGuDH7fQ/Lancelot/.

1 INTRODUCTION

Graphics Processing Units (GPUs) have demonstrated immense potential in accelerating database analytics due to their high memory bandwidth and massive computational power. Efforts in both academia [16, 37, 38, 49, 61] and industry [2, 4, 6, 8, 14, 15] have consistently shown that GPU DBMSes can achieve remarkable speedup ranging from 10–25× over CPU DBMSes. As modern GPU hardware rapidly evolves, the performance gap between CPUs and GPUs continues to widen, potentially leading to even greater speedup in the future. For instance, over the past three years alone, GPU peak performance and memory bandwidth have increased by 5× and 3.5× respectively [10].

While GPUs present a promising opportunity for database acceleration, they frequently encounter a limitation: the dataset often exceeds the capacity of GPU device memory, which ranges up to 192 GB [1] as of today. To address this challenge, existing research has adopted several distinct strategies. One effective strategy is to leverage CPU memory to support larger data sets and execute the query with both CPU and GPU (i.e., *heterogeneous CPU-GPU query processing*) [16, 23, 34, 60]. Such a design can exploit the parallelism of both CPU and GPU while minimizing the data transfer overhead between the two devices. Another strategy is to leverage multiple GPUs (i.e., *multi-GPU query processing*) [6, 27, 44, 46, 53]. Multi-GPU systems can offer larger aggregated total GPU memory and more computational power, leading to a bigger performance gain compared to a single GPU system.

The solutions developed in prior works, however, only apply for either hybrid CPU-GPU DBMS with a single GPU [16, 34, 41, 60] or strictly multi-GPU DBMS without the use of CPUs [44, 46, 53]. While some prior systems support both CPUs and multiple GPUs [6, 23, 36], none of these systems fully explore the design space produced by the *heterogeneity* and *distributive* nature of this architecture. For example, HetExchange [23] and HERO [36] do not have a data placement strategy across CPU and multiple GPUs. HeavyDB [6] resorts to a naive data placement strategy by simply partitioning the most recently used data across all GPUs.

In this paper, we aim to address the unique challenge of scaling heterogeneous CPU-GPU DBMS to multiple GPUs. One key challenge in this design space arises from the dual requirements of both *heterogeneity* and *scalability*. To address this challenge, we propose the *Unified Multi-GPU Abstraction*. This abstraction aims to simplify the design space by treating multiple GPUs as a single large monolithic GPU (shown in Figure 1). By doing this, we can decouple the design into a two-step process. The first step navigates the design space between CPUs and the Unified Multi-GPU, and the second step navigates the design space across multiple GPUs. Building on this abstraction, we develop **Lancelot**, a heterogeneous CPU and multi-GPU DBMS. Lancelot aims to scale the following critical design aspects in CPU-GPU DBMS to multiple GPUs:

Data placement. Leveraging both CPU and multi-GPU presents us with the opportunity to (1) cache the data collectively across multiple GPUs and (2) replicate data across GPUs to reduce communication. Both caching and replication can lead to better query performance since they can improve GPU utilization and minimize data transfer between devices. One unique challenge for data placement in this architecture is the coordination between caching and replication. These two goals may be in conflict, e.g., more data replicated across GPUs means less data can be cached in total in GPUs. To achieve the best overall performance, Lancelot introduces the *cache-aware replication policy*, a cost-based replication strategy

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704977

that selectively replicates shuffle-intensive data to strike a balance between caching and replication.

Query execution. Another challenge in scaling a heterogeneous DBMS to multiple GPUs is to transform the heterogeneous query plan to leverage multi-GPU hardware. Previous work such as Mordred [60] introduces *segment-level query execution*, which allows different segments of a column to execute different query plans depending on whether the segments are cached in GPU. This approach, however, does not scale to multiple GPUs, because the number of subquery plans will blow up due to the fact that segments can be located across multiple GPUs. In Lancelot, we solve this challenge by extending the *segment-level query execution* with distributed query processing techniques to run efficiently on multiple GPUs.

Overall, this paper makes the following contributions:

- We develop the *unified multi-GPU abstraction*, which views multi-GPU devices as a single large monolithic GPU to reduce the design complexity in hybrid CPU and multi-GPU DBMS.
- We develop a *cache-aware replication policy* for heterogeneous CPU and multi-GPU DBMS. The policy takes into account the cost of shuffling when replicating data and could coordinate both caching and replication decisions for the best performance.
- We extend *segment-level query execution*, a fine-grained hybrid execution strategy to support query plans on multiple GPUs.
- We build Lancelot, a hybrid CPU and multi-GPU analytical engine that incorporates the proposed optimizations. Our detailed evaluation shows that *cache-aware replication* can lead to 2.5× speedup and Lancelot can outperform existing GPU DBMSes by at least 2× on SSB and 12× on TPC-H.

The rest of the paper is organized as follows: We discuss the background in Section 2. Section 3 describes the *unified multi-GPU abstraction*. Section 4 describes the *cache-aware replication policy*. Section 5 describes our hybrid CPU and multi-GPU query execution strategy and its optimizations. Section 6 describes Lancelot's implementation details. Section 7 evaluates the performance of Lancelot. Section 8 discusses related work and Section 9 concludes the paper.

2 BACKGROUND

In this section, we describe the GPU architecture and previous works that support query execution on GPUs.

2.1 GPU Architecture

A GPU comprises a hierarchy of memory components and a collection of streaming multiprocessors (SMs). At the bottom of the hierarchy is the global memory which is often implemented using the high-bandwidth memory (HBM). Modern GPUs can boast up to 192 GB of global memory capacity, offering bandwidth reaching 5.2 TB/s [1]. Each SM, serving as the basic compute unit, features a fixed set of registers and a shared memory accessible by all cores within the SM. Global memory accesses can be cached in the L1 cache local to each SM or the shared L2 cache across all SMs.

In GPU programming models [3, 7, 12, 54], threads are organized into thread blocks, typically comprising 32 to 1024 threads, each executed by a single SM. Threads within the same thread block can synchronize and share data using the shared memory. Thread blocks are further divided into groups of 32 threads, known as warps, which execute instructions following the Single Instruction Multiple Threads (SIMT) model. Accesses to neighboring memory addresses by a warp are coalesced into a single memory transaction.

2.2 Data Analytics on GPUs

A flurry of existing works in academia [25, 26, 38, 43, 48, 57, 61] and commercial systems (e.g, HeavyDB [6], PG-Storm [13], cuDF [4], BlazingSQL [2]) have attempted to accelerate query execution on GPUs. Some prior works focused on accelerating individual operators such as selection [51], join [30, 32, 35, 39, 47, 50, 58], sort [52], and user-defined functions [59]. There have been works that offer a complete set of database operations [4, 6, 31, 48, 49] but often suffer from limited GPU memory capacity. For example, Crystal [49] and cuDF [4] only support workloads that fit in the GPU memory.

To mitigate this limitation, one common approach is to store the complete data set on CPUs [6, 37, 48, 61] and execute the query in multiple stages on GPU. For example, YDB [61] and HippogriffDB [37] stream the compressed data from CPU memory and execute one operator at a time. HeavyDB [6] will cache only the most recently used data in GPU and transfer the rest of the data from the CPU on-demand during query execution. Even though such systems no longer suffer from limited GPU memory capacity, the data transfer overhead between CPU and GPU will be the performance bottleneck due to the limited interconnect bandwidth.

2.3 Heterogeneous CPU-GPU DBMS

To minimize data transfer while supporting datasets larger than GPU memory, existing works have attempted to use both CPU and GPU for query execution [16, 23, 29, 33, 34, 36, 40, 41, 60, 62, 63]. By partially executing the query on CPU, excessive data transfer can be reduced. For example, CoGaDB [16] and Ocelot [34] cache hot columns in the GPU memory and use a cost-based optimizer [17, 19–21] to assign operators to either CPU or GPU. HetCache [41] caches NVMe resident data on CPU and GPU based on query processing throughput and selectivity information. It caches densely accessed pages in GPU memory and sparsely accessed pages in CPU memory.

Finally, Mordred [60] explores the design space of data placement and heterogeneous query execution for in-memory CPU-GPU DBMS. It introduces the *semantic-aware caching policy* which takes into account query semantics, data correlation, and query frequency when determining data placement between CPU and GPU. It also introduces the *segment-level query execution* — a query executor that can fully exploit data in both devices and coordinate query execution at a fine granularity. Mordred [60] has been shown to outperform the other CPU-GPU DBMSes by 11×.

However, despite showing notable performance gains, these solutions only support a single GPU device, missing the speedup opportunity of multiple GPUs. Lancelot is built on top of Mordred with the focus on scaling these solutions to multiple GPUs.

3 UNIFIED MULTI-GPU ABSTRACTION

One key challenge in expanding a heterogeneous DBMS to multiple GPUs arises from the need to address both *heterogeneity* and *scalability*. In Section 3.1, we describe the challenges and discuss the limitations of previous work. Section 3.2 describes our proposed solution, *unified multi-GPU abstraction*, which reduces the design complexity by treating multiple GPUs as a single large GPU.

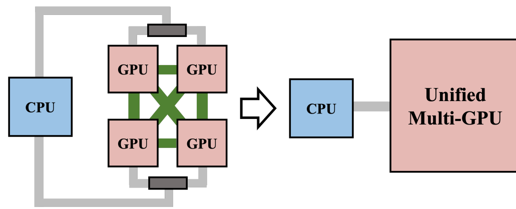


Figure 1: Illustration of Unified Multi-GPU Abstraction

3.1 Challenge

In hybrid CPU and multi-GPU DBMS, one key challenge stems from the need to navigate both *heterogeneity* (CPU vs GPU) and *scalability* (across multiple GPUs) when making design decisions. Such requirements can significantly escalate the design complexity. Prior solutions focus on only one aspect but have not thoroughly explored the design space of data placement and query execution in hybrid CPU and multi-GPU DBMS. For example, HetExchange [23] provides a framework to express hybrid query plans but does not explore the design space of data placement and query execution in CPU and multi-GPU DBMS. HERO [36] explores the design space for operator placement but does not address the challenges in other database aspects, such as data placement and query execution.

3.2 Design Abstraction

To simplify the design space in hybrid CPU and Multi-GPU DBMS, we introduce the *Unified Multi-GPU Abstraction*. The main idea behind this abstraction is to view multiple GPUs as a unified, large monolithic GPU, rather than as individual discrete GPUs. Figure 1 illustrates this abstraction. With the unified multi-GPU abstraction, the system design is decomposed into two separate steps:

Step 1: Addressing heterogeneity. In this step, we treat multiple GPUs as a single GPU with larger aggregated memory and more processing power. For the case of data placement, the DBMS determines what data should be cached in the *unified Multi-GPU* with respect to the CPU, but does not concern data placement across the GPUs. By doing this, we can directly apply existing techniques that have been developed for heterogeneous CPU-GPU DBMS.

Step 2: Addressing scalability. In the second step, we zoom inside the *unified Multi-GPU* to address the challenges posed by multiple GPUs. In particular, the DBMS will decide how the cached data should be partitioned and/or replicated across the GPUs.

The *unified multi-GPU abstraction* is tailored for systems with homogeneous GPUs, which is common in data centers and the cloud. Moreover, modern GPUs often use fast interconnects like NVLink, capable of up to 450 GB/s [11], making inter-GPU data transfer significantly faster than CPU-to-GPU transfer.

In Lancelot, we use *the unified multi-GPU abstraction* as a guideline to explore the design space in hybrid CPU and multi-GPU DBMS. This abstraction is straightforward and does not require additional formulation. In the next two sections, we will discuss in detail the implementation of this abstraction for data placement (Section 4) and query execution (Section 5).

4 DATA PLACEMENT

Intelligent data placement can lead to better query performance and memory efficiency in heterogeneous CPU-GPU DBMSes. Lancelot treats data placement as a caching problem following previous works [16, 60] — the complete data set resides in CPU memory and a mirrored subset of data is cached in GPUs. Lancelot borrows the *semantic-aware fine-grained caching policy* [60], which is the state-of-the-art caching policy in heterogeneous CPU-GPU DBMS.

To address data placement across multiple GPUs, Lancelot treats data placement as a replication problem inspired by previous works in distributed databases [55] — the data set can be either partitioned and distributed across multiple GPUs or replicated across multiple GPUs. Whether a piece of data should be replicated is determined by the benefit of replication (e.g., reduction of network transfer) and the associated cost (e.g., extra disk/memory space consumption).

A unique challenge in hybrid CPU and multi-GPU is the correlation between (1) the caching policy in the *unified Multi-GPU* and (2) the replication policy across GPUs. More data replication across GPUs means less data can be cached in total. To achieve the best overall performance, the caching and replication decisions must be holistically considered to balance their effects on performance.

Lancelot addresses this challenge by developing a *cache-aware replication policy*. The key insight behind the policy is to use a unified cost model to estimate the effects of caching and replication. In Section 4.1, we demonstrate how replication can lead to better performance. Then, we explain the proposed policy in Section 4.2.

4.1 Motivation

4.1.1 *The benefit of shuffle-awareness.*

In a multi-GPU system, despite being interconnected with high-speed interfaces like NVLink, data transfer between GPUs can still be the bottleneck of query execution since NVLink bandwidth is lower than that of the GPU device memory. Existing multi-GPU DBMSes always partition the data across multi-GPUs (Figure 2a) which often results in significant portions of query execution spent on shuffling data across multiple GPUs [5, 6, 44]. Our experiments show that when running a co-partitioned join with 4 V100 GPUs connected by NVLink, around 50% of the total join runtime is consumed by partitioning and transferring data across GPUs.

By strategically replicating data, we can reduce or even eliminate the data shuffling overhead. Figures 2a and 2b illustrate the benefit of shuffle-aware data replication. Without data replication (Figure 2a), joining relations R and S requires the DBMS to either (1) broadcast column Z (which is a join key) or (2) co-partition R and S on columns Y and Z, respectively. Figure 2b shows both GPUs' cache content after *shuffle-aware data replication*. By replicating $Z0$ and $Z2$, we now only need to broadcast $Z1$ and $Z3$ to perform the join locally in each GPU. Compared to Figure 2a, the data transfer overhead during join is now halved. Our experiment running join with 4 V100 GPUs shows that replication can lead to 2× speedup.

4.1.2 *The benefit of cache-aware replication.*

As depicted in Figure 2b, data replication will consume the available cache size in each GPU. As replication continues, we will reach the limit of GPU memory capacity, bringing us to a decision point where we can either (1) stop replication or (2) opt to continue replication at the cost of evicting some data back to the CPU.

Continuing replication at the expense of evicting cached data may result in worse query performance since it may lead to certain query operations executed on CPUs. Conversely, it may also lead to speedup, particularly if the evicted data are infrequently accessed

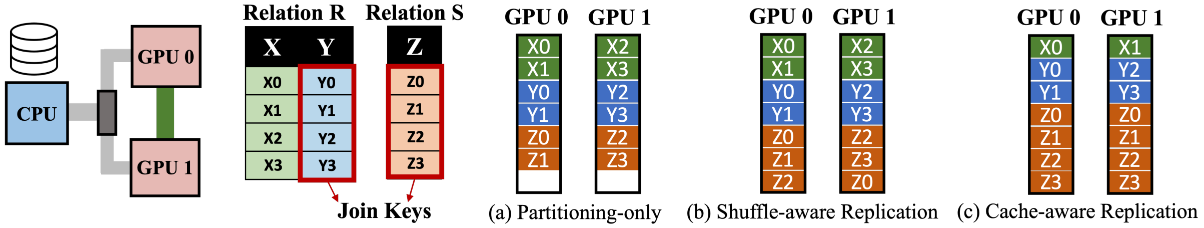


Figure 2: Illustration of Cache-aware Replication Policy.

Algorithm 1: Update the replication weight for Segment S

```
   # estimate the cost of shuffling
1  Cost = estimateShuffleCost(shuffled_segments)
2  for S in shuffled_segments do
       # increment replication weight of segment S
3      S.replication_weight += Cost / (|shuffled_segments| × |CARD|)
```

Algorithm 2: Decide between Caching and Replication

```
1  CacheOrReplicate(segment S_x, segment S_y):
2  if S_x.replication_weight > S_y.caching_weight then
3      for gpu in NUM_GPU do
4          Cache(S_x, gpu) # Replicate S_x
5  else
6      gpu = mapSegmentToGPU(S_y) # Hash function to select gpu
7      Cache(S_y, gpu)# Cache S_y in gpu
```

(cold data) or involved in operations that can be executed efficiently on CPUs. This underlines the importance of integrating cache-awareness into the replication policy to effectively navigate the trade-off between caching and replication. Such property has not been exploited by existing works.

Figure 2c illustrates the benefit of *cache-aware* data replication. In this example, column X is only used for aggregation, which does not introduce much overhead when executed on CPUs. Therefore, the policy chooses to replicate data from column Z instead of caching data from column X. Doing this allows the join to be performed locally in each GPU at the cost of running aggregation on the CPU.

4.2 Cache-Aware Replication Policy

The key challenge of a hybrid CPU and multi-GPU DBMS is to find a holistic solution to handle both caching and replication decisions. Replication reduces data transfer between GPUs but also results in caching less data, hence executing more operators on the CPUs. In contrast, caching would ensure more operators to be executed on the GPUs, but may result in excessive data transfer across multiple GPUs. As both caching and replication would lead to performance speedup and space consumption in the GPU memory, we need to resolve the conflict between the two policies to achieve the best overall performance. The goal of our *cache-aware replication policy* is to: (1) selectively replicate data to reduce data transfer overhead, and (2) navigate the trade-off between caching and replication.

The key insight of our policy is to use a unified cost model to estimate the effect of both caching and replication. Our policy splits each column into equal-sized partitions, which we will refer to as *segment* for the rest of the paper. The policy will cache and replicate data on segment granularity. Each data segment will be assigned weights derived through cost models which will be used to navigate the trade-off between caching and replication. Section 4.2.1 describes the general replication framework and Section 4.2.2 describes the cost model used by the policy.

4.2.1 *Replication Policy.*

Our *cache-aware replication policy* is inspired by the *semantic-aware caching policy* used in Mordred [60]. In Mordred, each segment will be assigned a weight that reflects the benefit of caching the segment in GPU. The weight is derived using the cost model from Crystal [49] and data will be cached starting from the segment with the highest weight. Lancelot uses the same weight-based mechanism for caching data in GPU. Furthermore, Lancelot generalizes the mechanism to support both *caching* and *replication*.

Algorithm 1 shows the shuffle-aware weight update in our replacement policy. We execute Algorithm 1 each time we shuffle or broadcast data during query execution. The algorithm first predicts the data shuffling cost using the function *estimateShuffleCost()*, which utilizes a simple cost model that will be described in detail in Section 4.2.2. Following this, for each segment that is involved in data shuffling or broadcast, we will increment the replication weight by the data shuffling cost divided by the total number of segments involved and normalized by the table cardinality.

Since both caching and replication policies in Lancelot collect per segment weight, each segment now possesses two weighted counters: (1) replication weight from the *cache-aware replication policy* and (2) caching weight from the *semantic-aware caching policy*. Both policies derive their weights from the same cost model [49], so that direct comparison between the two weights becomes feasible. Function *CacheOrReplicate()* outlined in Algorithm 2 can be used to determine whether to replicate Segment S_x or cache Segment S_y in order to occupy the available GPU cache space: if the replication weight of S_x is larger than the caching weight of S_y, we opt for replication of S_x; conversely, if the caching weight of S_y surpasses the replication weight of S_x, we cache S_y instead.

For example, in Figure 2, segments from column X have low caching weights since they are only used for aggregation, which does not impose much overhead when executed on CPUs. In contrast, segments from columns Y and Z have high caching and replication weights since joins are expensive and may result in excessive data transfer across the GPUs. Given that the replication weight of column Z is higher than the caching weight of column X, Algorithm 2 would decide to replicate Z rather than caching X to occupy the available GPU memory space.

Since our policy requires prior knowledge of query statistics, Lancelot gathers these statistics and calculates the weight after each run. Caching and replication are performed periodically every n queries, with n being a user-defined parameter set to 100 by default.

4.2.2 *Cost Model.*

This subsection explains how the *estimateShuffleCost()* in Algorithm 1 works. In particular, we use the cost model presented in Crystal [49] and Mordred [60] to estimate the cost of shuffling or broadcasting the data across GPUs. These models operate under the assumption that queries can fully utilize memory bandwidth, thereby deriving execution time from memory traffic. The accuracy of the model has been verified in Crystal on simple operators. Mordred extends the model to support hybrid CPU-GPU execution and PCIe, demonstrating acceptable accuracy for the purposes of caching policy. In this work, we extend the model even further to support operators used in multi-GPU query execution.

In particular, in distributed query execution with multiple GPUs, there is an extra cost for data transfer and data partitioning. We model the cost of data transfer as follows:

$$data_transfer = \frac{size(int) \times N}{BW_{gpu2gpu}}$$

Where N is the cardinality of input segments and $BW_{gpu2gpu}$ is the interconnect bandwidth between GPUs. We model the cost of data partitioning as follows:

$$data_partitioning = \frac{size(int) \times N}{B_r} + \frac{(1 - \pi) \times N \times C}{B_w}$$

Where B_r and B_w are GPU read and write memory bandwidth, C is the cache line size, and π is the probability the accessed cache line is in the last level cache. Using these equations as building blocks, we can express various communication collectives in *estimateShufleCost()*. For example, *broadcast* and *all-to-all* can be expressed with a collection of *data_transfer* between GPUs as follow:

$$broadcast = \sum_{i=1}^{nGPUs} \frac{size(int) \times N}{BW_{gpu2gpu}}$$

$$all\text{-}to\text{-}all = \sum_{i=1}^{nGPUs} \sum_{j=1}^{nPartition} \frac{size(int) \times N_{ij}}{BW_{gpu2gpu}}$$

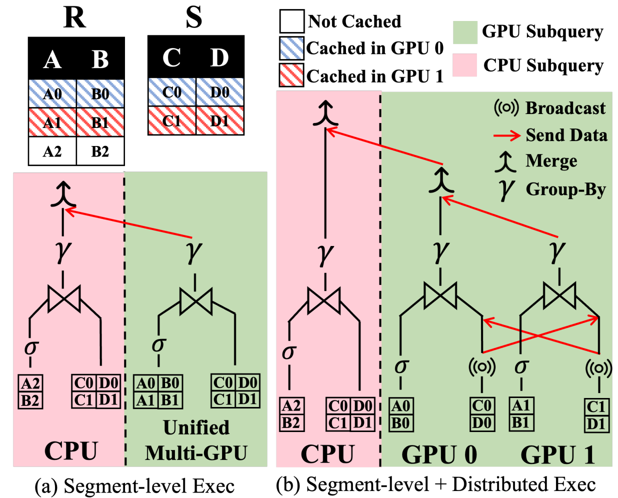
Similarly, data shuffling can be expressed as *data_partitioning* on each GPU followed by an *all-to-all communication*.

Since our cost model takes into account the hardware properties (e.g. interconnect bandwidth, GPU memory bandwidth, etc.), our policy can adapt to various hardware and interconnect speeds.

5 QUERY EXECUTION

Scaling heterogeneous query execution to multiple GPUs introduces new challenges and opportunities. First of all, since caching and replication are performed in segment granularity, we need to maintain fine-grain coordination during query execution across the CPU and multiple GPUs. Previous work has attempted to address such challenge with *segment-level query execution* — different segments of a column will execute different subquery plans depending on the segments' location — but does not extend the solution for multiple GPUs. In Lancelot, we leverage the *unified multi-GPU abstraction* to extend *segment-level query execution* to multiple GPUs.

Moreover, as discussed in Section 4.1.1, multi-GPU query execution can introduce significant data transfer overhead. In Lancelot, we further minimize this overhead by applying several optimizations from distributed query processing to multi-GPU query execution. Section 5.1 describes the general query execution framework in Lancelot and Section 5.2 describes various optimizations in Lancelot to speedup the query performance.



(a) Segment-level Exec (b) Segment-level + Distributed Exec

Figure 3: Example of Query Execution in Lancelot

5.1 Scaling Query Execution to Multiple GPUs

By leveraging the *unified multi-GPU abstraction*, we can decouple query execution into two steps: (1) query execution between CPU and the *Unified Multi-GPU*, and (2) query execution across multiple GPUs. We will discuss how Lancelot addresses each step individually in Section 5.1.1 and Section 5.1.2. Then, we show an example of query execution in Lancelot in Section 5.1.3.

5.1.1 *Query Execution between CPU and the Unified Multi-GPU.*

One unique challenge in query execution involving both CPU and multiple GPUs lies in the communication between the two device types. Lancelot covers 4 different data transfer scenarios:

Transferring different partitions from CPU to GPUs: In this scenario, data is partitioned on the CPU before each partition is sent to different GPUs. Lancelot will utilize all available PCIe lanes to transfer the data from the CPU to each GPU efficiently.

Broadcasting data from CPU to GPUs: For NVLink-connected GPUs, Lancelot will send the data to one GPU, which then broadcasts the data to all the other GPUs. Broadcasting data from the GPU is typically faster due to the high-speed NVLink interconnect. For PCIe-connected GPUs, Lancelot will utilize all available PCIe lanes to broadcast the data from the CPU to each GPU.

Transferring groupby/aggregation results from GPUs to CPU: Since merging results in GPU is faster than CPU, Lancelot will first merge the groupby result in one GPU, before sending it to the CPU.

Transferring join/filter results from GPUs to CPU: In this case, Lancelot will use all the available PCIe lanes to directly transfer the different partitions from each GPU to the CPU.

To coordinate fine-grain query execution across CPUs and GPUs, Lancelot adopts the *segment-level query plan* [60]. *Segment-level query plan* allows different segments of a column to execute different query plans depending on the segments' location. Lancelot uses a simple heuristic to push an operation to where the segments reside to minimize the data transfer. Figure 3a shows an example of a segment-level query plan. To support multi-GPU query execution, Lancelot will simply expand each GPU subquery plan to a *distributed query plan* (Figure 3b). Doing this will prevent the heuristic from generating an excessive number of subquery plans.

5.1.2 Query Execution across Multiple-GPUs.

Lancelot executes a multi-GPU query plan similar to how a distributed query engine works. For distributed join, Lancelot supports both broadcast and co-partitioning join. For filtering, Lancelot executes the operator locally in each GPU. For group-by and aggregation, Lancelot first executes the operator locally in each GPU, followed by merging the results across all the GPUs.

Lancelot provides three multi-GPU communication routines to support distributed query plans as follows:

Broadcast: This routine is used to send data from one GPU to all the other GPUs. Common use cases include broadcast join and broadcasting intermediate results from the CPU.

Gather: This routine is used to collect data from all the GPUs to one GPU. Mainly used for merging group-by results across GPUs.

All-to-all: This routine is used in co-partitioned join to exchange data between every pair of GPUs, allowing a GPU to send and receive its partition from every other GPU.

Lancelot implements these routines by leveraging NCCL [9], a topology-aware multi-GPU communication library that has been optimized to achieve high throughput on multi-GPU platforms.

5.1.3 Example of Query Execution.

```
Q0: SELECT S.D, SUM(R.A) FROM R,S
    WHERE R.B = S.C AND R.A > 10 GROUP BY S.D
```

Figure 3 illustrates an example of multi-GPU query execution in Lancelot when executing $Q0$. In this example, R is the probe relation and S is the build relation. In Figure 3a, we can see how the query plan is divided into subquery plans following the *segment-level query plan* based on where the data is originally located. Specifically, the query is divided into 2 subqueries:

CPU Subquery: This subquery operates on uncached data ($A2$, $B2$) and executes filter, join, and group-by on CPUs.

GPUs Subquery: This subquery operates on cached data ($A0$, $B0$, $A1$, $B1$) and executes filter, join, and group-by on GPUs.

Lancelot will launch both subqueries in parallel to utilize all the available computation power of CPUs and multiple GPUs.

Figure 3b shows the generated query plan after Lancelot expands the **GPU Subquery** into a *distributed query plan* for 2 GPUs. For simplicity, we will use broadcast join as our join strategy, although other approaches such as co-partitioning join can also be applied. To perform the broadcast join, segments $C0$, $C1$, $D0$, and $D1$ will be broadcasted such that each GPU has the complete copy of relation S. Doing this will allow all remaining operations (filter, join, group-by, and aggregation) to be executed locally on each GPU. Finally, after each GPU computes its final result, results from GPU 0 and GPU 1 are merged in GPU 0 before being transmitted to the CPUs for the final aggregation with the **CPU subquery**.

5.2 Reducing Data Transfer Overhead

In this section, we describe optimizations we adopt in Lancelot to further reduce the data transfer overhead across GPUs. We will discuss three optimizations: late materialization (Section 5.2.1), adaptive join (Section 5.2.2), and join reordering (Section 5.2.3).

5.2.1 Multi-GPU Late Materialization.

During query execution, transferring intermediate relations can often be expensive. For example, a co-partitioned join operation will partition either relation into multiple smaller partitions, and transfer each partition to the corresponding GPUs. Previous CPU-based columnar databases used the late materialization strategy to reduce data transfer by expressing intermediate relations in the form of row IDs. Lancelot employs a late materialization strategy specifically tailored for multi-GPU query execution.

During the partitioning phase in co-partitioning join, Lancelot will reconstruct only the join key columns used by the query. The remaining columns in the relation will be expressed in the form of row IDs. For subsequent query operations, Lancelot reconstructs the accessed columns using row IDs lazily, by leveraging the *Unified Virtual Addressing* (UVA). UVA is a GPU-specific technology that allows GPU kernels to directly access peer GPUs' memory in byte granularity through the device interconnect (e.g. NVLink).

For queries with highly selective joins, this reconstruction cost can be negligible, reducing the overall data transfer overhead. For queries with non-selective join, however, reconstruction can introduce non-trivial overhead due to random accesses to the peer GPUs' memory. Hence, to decide whether to do late materialization, Lancelot leverages a cost model to estimate the reconstruction overhead as follows:

$$reconstruction = \frac{N \times C}{BW_{gpu2gpu}}$$

Where N is the number of tuples to reconstruct and C is the cache line size. Based on this estimation, Lancelot will do late materialization if the reconstruction cost is cheaper than the cost of broadcasting or shuffling the input data (obtained from Section 4.2.2).

5.2.2 Adaptive Join Strategy.

Lancelot supports two types of distributed joins: broadcast join and co-partitioning join. Since both joins have different costs, choosing the right join strategy could lead to performance speedup. For example, when joining with small relations, broadcast join is often more efficient compared to co-partitioning join, since the broadcasting small relations incurs little overhead. To decide between the two strategies, Lancelot once again leverages the cost model in Section 4.2.2 to estimate the cost of both joins and choose the join strategy with a lower cost. While the model might not be precise, we find the accuracy to be acceptable for this purpose.

5.2.3 Join reordering.

We can further reduce the data transfer during query execution by reordering the joins based on their join strategy. In particular, if we have an n-way joins with a left-deep join tree query plan, Lancelot will reorder the co-partitioning joins to the end of the join pipeline. Executing co-partitioning join at the start of the join pipeline can be expensive since we would need to shuffle the full input relations that are relatively large. By pushing the joins to the end of the pipeline, we can speed up the co-partitioning joins as the sizes of the input probe relations would often have been reduced by the selectivity of the prior joins in the pipeline.

6 SYSTEMS INTEGRATION

This section describes the implementation of Lancelot, our hybrid CPU and multi-GPU DBMS. Figure 4 illustrates the architecture of Lancelot, which consists of three main modules described in the following subsections: Buffer Manager (Section 6.1), Query Optimizer (Section 6.2), and Query Execution Engine (Section 6.3).

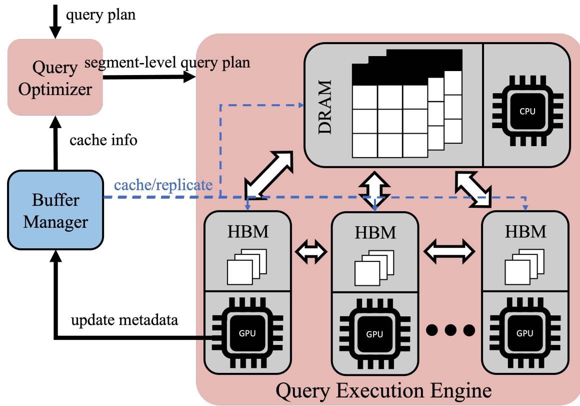


Figure 4: Lancelot System Architecture

6.1 Buffer Manager

The Buffer Manager is a CPU component that manages data placement on the CPU and across multiple GPUs in Lancelot. It divides each GPU memory into two regions:

Data Caching: This is a region that stores the cached or replicated data in segment granularity. It performs periodic data placement by leveraging *semantic-aware caching* and *cache-aware replication policy* discussed in Section 4. The segment size is user defined and set to 4MB by default.

Data Processing: This is a region which stores intermediate results during query execution (i.e., hash tables, intermediate results, etc.). Both the data caching and the processing region are pre-allocated with a fixed size during system initialization to avoid frequent dynamic memory allocation in the GPUs.

The Buffer Manager also handles the metadata management in Lancelot. The two main data structures used by our buffer manager are an array of free lists (one free list per GPU) and a 2-level hash table. We use the free list in each GPU to track the available slots in their respective caches. We use the 2-level hash table to store the location of each segment. The first level maps each segment to the corresponding GPU, and the second level maps to an address in the GPU memory. Although using a single-level hash table is also possible, we opt for 2-level hash table due to its simplicity, especially since metadata management has a negligible impact on query performance. The Buffer Manager also stores the segment statistics in Lancelot, such as the replication weight, caching weight, and the min-max of each segment.

The metadata of Lancelot resides in the CPU memory. When a GPU kernel requires some metadata, the Buffer Manager will send the necessary metadata to the GPUs prior to launching the kernel.

6.2 Query Optimizer

The query optimizer converts a query plan to the segment-level query plan shown in Figure 3(b). Our original query plan is taken from Crystal [49] which is already optimized for GPUs.

To decide which operator should be executed on GPUs, our optimizer leverages the *data-driven operator placement*[18], where we execute operators on GPUs only when the input data is cached in at least one of the GPUs. Subsequently, to construct the multi-GPUs query plan, the optimizer will replace the join with either broadcast join or co-partitioning join and apply the optimizations in Section 5.2.2 and Section 5.2.3. For each group-by and aggregation, the optimizer will insert a merge operator following the group-by.

6.3 Query Execution Engine

The Query Execution Engine in Lancelot executes the query plan generated by the query optimizer across CPU and multiple GPUs. In particular, similar to Mordred, it executes each subquery plan in parallel and merges the results at the end.

CPU execution in Lancelot is implemented with Intel TBB. GPU execution in Lancelot is implemented with CUDA, using Crystal library [49]. Lancelot extends Crystal to support multi-GPU query execution, such as distributed join and merge operations. Communication across multiple GPUs is implemented using NCCL [9] and the `cudaMemcpy()` primitives in CUDA. To launch a multi-GPU kernel, Lancelot switches the device context to each GPU using `cudaSetDevice()` API and launch the kernel asynchronously in each GPU by leveraging the `cudaStream_t` primitives.

7 EVALUATION

In this section, we evaluate the performance of Lancelot. The section will answer the following key questions:

- How does Lancelot scale as we increase the number of GPUs?
- How does the *cache-aware replication policy* perform compared to other data placement policies?
- How much performance improvement is achieved through various optimizations introduced in Section 5.2?
- How does Lancelot perform compared to other GPU DBMSs?

7.1 Experimental Setup

Hardware configuration: We ran our experiment with three different compute instances:

- **GPU3.8 instance (OCI):** This instance features 8 NVIDIA V100 GPUs connected through NVLink 2.0 with up to 300 GB/s bidirectional bandwidth per GPU. Each V100 GPU has 16 GB of HBM2 memory with a read/write bandwidth of 880 GBps. These GPUs are connected to 52-Cores Intel® Xeon® E5-2698 via PCIe3 with 12.8 GB/s bidirectional bandwidth.
- **GPU4.8 instance (OCI):** This instance features 8 NVIDIA A100 GPUs connected through NVLink 3.0 with up to 600 GB/s bidirectional bandwidth per GPU. Each A100 GPU has 40 GB of HBM3 memory with a read/write bandwidth of 1550 GB/s. These GPUs are connected to 64-Cores AMD EPYC 7542 (Rome) CPUs via PCIe4 with 25.6 GB/s bidirectional bandwidth.
- **g6.48xlarge (AWS):** This GPU instance features 8 NVIDIA L4 GPUs paired with 96-Cores AMD EPYC 7R13 (Milan) CPUs. Each L4 GPU has 24 GB of GDDR5 memory with a read/write bandwidth of 320 GB/s. The GPUs and CPUs are connected via PCIe4 with 25.6 GB/s bidirectional bandwidth. Since there is no NVLink, GPU-to-GPU communication also occurs through PCIe.

Benchmark: Most of our experiments use the *Star Schema Benchmark* (SSB) [42]. The SSB dataset has five tables with one fact table and four dimension tables. Since the dimension tables in SSB are very small ($< 0.5\%$ of the fact table), we increase the size of the dimension table to demonstrate more interesting scenarios when comparing different data placement policies. The supplier, customer,

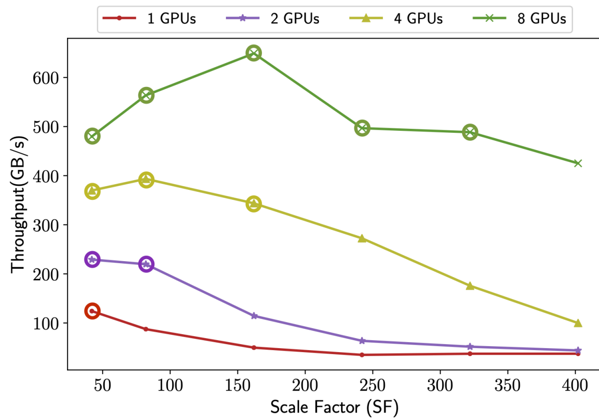


Figure 5: Performance Throughput of Lancelot with Different Scale Factors and Number of GPUs

and part tables are now 5%, 15%, and 25% of the lineorder table size respectively. In Section 7.5, we will also run the experiments with a subset of the TPC-H benchmark. We follow the method described in [56] to select a representative subset of queries.

To enable efficient query execution in GPU, we store the data in columnar stores and dictionary encode the string columns into integers prior to data loading. Therefore, we ensure that all column entries are 4-byte in value. In our evaluation, the entire data set is loaded to CPU memory before each experiment starts.

Measurement: Unless otherwise stated, we dedicate 50% of each GPU memory for data caching, and the other half for data processing. Before each experiment, we first warm up the GPU memory by running 100 random queries and then perform the caching and replication to populate the GPU memory. For each experiment, we will run 100 random queries and measure the query execution time.

7.2 Scalability Evaluation

This subsection evaluates how Lancelot can scale to multiple GPUs. In this experiment, we use a GPU3.8 instance and sweep the scale factor from 40 to 400, measuring throughput when running 100 random SSB queries. $SF = 40$ translates to a total of 7.5GB of data that are accessed by all the SSB queries (fits in a single GPU cache). The data size of other scale factors (80–400) is a multiple of $SF = 40$. In this experiment, the throughput is calculated as the total amount of input data scanned by the queries divided by the total query execution time. Figure 5 shows the result of the experiment when running with 1, 2, 4, and 8 GPUs. *The circles around the data points indicate that the data still fits in the aggregated GPU memory.*

Overall, as the data size increases on the x axis, the throughput decreases especially when the data can no longer be replicated (more data transfer overhead) or no longer fits in the aggregated GPU memory (queries are partially executed on the CPU). For example, when running with 1, 2, and 4 GPUs, data exceeding $SF = 40$, $SF = 80$, and $SF = 160$ respectively, no longer fits in the GPU memory, leading to diminished performance. The degradation, however, can be kept minimum due to our intelligent data placement policy.

Figure 5 also shows that increasing the number of GPUs does not always scale throughput linearly. For example, the throughputs of 2, 4, and 8 GPUs for $SF = 40$ are only 1.85×, 3×, and 3.85× higher than a single GPU. This is because when running $SF = 40$ on 4 and 8 GPUs, each GPU only processes a small amount of the data, leading to underutilization. In Section 7.5.1, we will show an experiment investigating the scalability when each GPU is fully utilized.

For 8 GPUs, the throughput increases up to $SF = 160$ and then declines. At $SF = 40$ and $SF = 80$, using 8 GPUs for query execution results in GPU underutilization, which limits the throughput. As the data size increases, throughput improves, peaking at $SF = 160$ when each GPU is fully utilized. At $SF = 240$ and $SF = 320$, Lancelot can no longer fully replicate the dimension tables, leading to increased data transfer and reduced throughput. Finally, at $SF = 400$, the data no longer fits in GPU memory, further lowering throughput as queries will be partially executed on the CPU.

7.3 Comparisons between Different Data Placement Policies

This subsection evaluates the performance of the *cache-aware replication policy* against other data placement policies. In this experiment, all compared policies will use the *semantic-aware caching policy* as their caching policy. We will then compare the performance of three different replication policies:

- **Replication-Only:** This policy will replicate all the segments that are cached across all the GPUs.
- **Partitioning-Only:** This policy will partition all the segments that are cached across all the GPUs. Segments are assigned to different GPUs in a round-robin fashion.
- **Cache-Aware Replication:** The *cache-aware replication policy* described in Section 4.

7.3.1 *Performance on Standard SSB.*

In this experiment, we sweep the number of GPUs and measure the query execution time of different data placement policies when running 100 random SSB queries. We run each experiment on three different GPU instances: GPU3.8 (V100), GPU4.8 (A100), and g6.48xlarge (L4); For each configuration, we will use 8GB cache size per GPU. We sweep the number of GPUs from 1 to 8 and run the experiment with $SF = 80$, $SF = 160$, and $SF = 320$. Therefore, all columns that are accessed by queries fit in 2 GPUs for $SF = 80$, 4 GPUs for $SF = 160$, and 8 GPUs for $SF = 320$. The query access distribution is uniform following the default configuration.

Figures 6(a)–6(c) show the result of our experiment. For V100 GPUs (top) and A100 GPUs (center), *Partitioning-Only* performs better than *Replication-Only*. On the other hand, for L4 GPUs (bottom), *Replication-Only* performs better than *Partitioning-Only*. *Cache-Aware* outperforms the other policies in all the cases.

Replication-Only fails to efficiently leverage the combined GPU capacity due to the necessity of replicating each segment across all GPUs. While this approach eliminates the data transfer overhead between GPUs, it also minimizes data caching within the GPU memory. Consequently, query execution primarily occurs on the CPU, leading to performance degradation.

Partitioning-Only can cache the most data in GPUs since none of the data will be replicated. As a result, it can execute a bigger portion of queries on GPUs compared to the other two policies. However, the performance may be suboptimal since fully partitioning data across the GPUs could incur significant data transfer overhead. This has been particularly evident with L4 GPUs (bottom). Since these

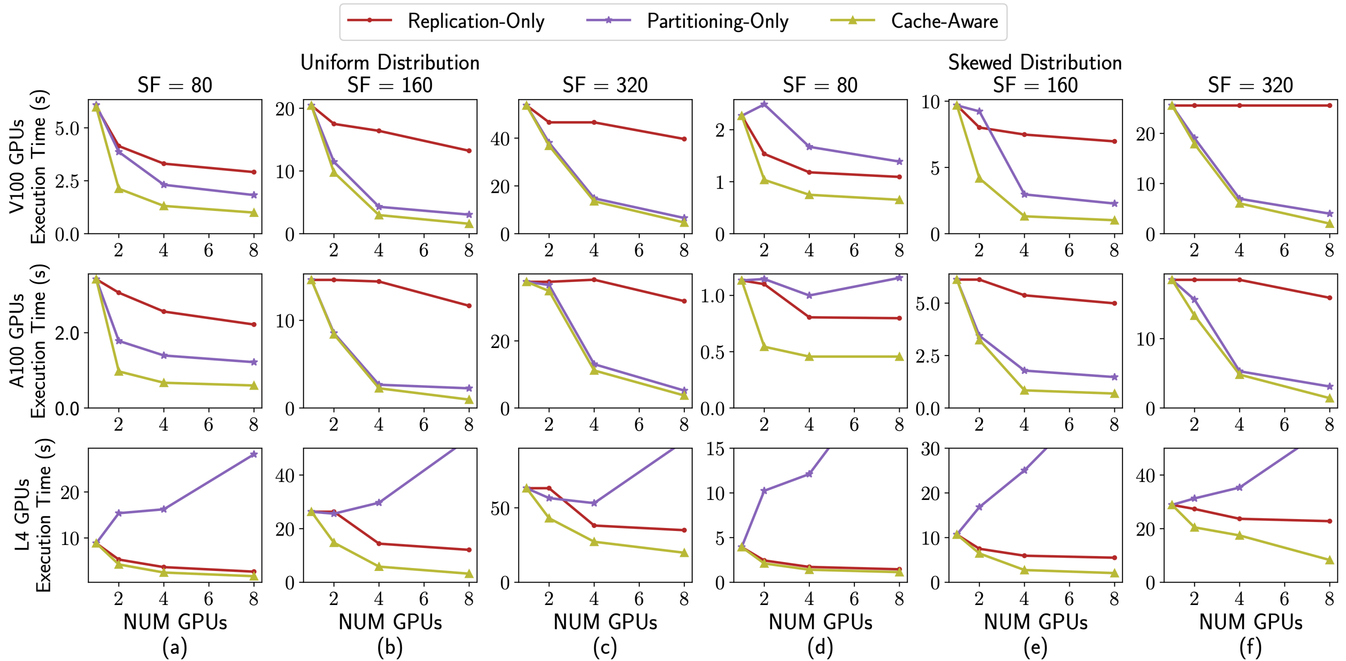


Figure 6: Execution Time of Different Data Placement Policies with Uniform (a–c) and Highly-Skewed (d–f) Distribution.

GPUs do not feature NVLink, inter-GPU communication occurs through PCIe, causing significant performance degradation. On V100 and A100 GPUs, this policy performs better as transferring data is cheaper due to the fast NVLink interconnect.

Our *Cache-Aware replication* outperforms the other policies in all cases by up to 2.5×. Our policy will replicate segments that give the most replication benefit which leads to speedup compared to *Partitioning-Only*, but stops replication if it starts to hurt caching performance. Thus, it can better utilize the GPU memory capacity while minimizing the data transfer overhead across GPU devices.

7.3.2 *Performance on Skewed Workload.*

This experiment evaluates the performance of *cache-aware replication policy* under highly skewed query access distribution. To simulate nonuniform distribution, we incorporate skewness into the date predicates of SSB queries such that more recent data has a higher probability of being accessed by the query. We pick the values following a Zipfian [28] distribution, resulting in 90% of the queries accessing the data from the last 3 years. Figures 6(d) - 6(f) show the results of the experiment.

Overall, *Cache-Aware Replication* outperforms the other policies across all scale factors and hardware configurations by up to 3×. *Replication-Only* performs better on the skewed workload than on the uniform workload. This is because caching only the hot data can already give us a decent performance speedup for highly skewed workloads, leaving more GPU memory capacity to benefit from replication. In fact, despite the faster NVLink interconnect on V100 (top) and A100 (center) GPUs, *Replication-Only* can outperform *Partitioning-Only* at smaller scale factors (SF=80 and SF=160).

We observe more speedup compared to the uniform distribution. This advantage stems from the policy's adept handling of skewed workloads, prioritizing the replication of segments from dimension tables over caching colder data from the fact table. This strategic decision yields more substantial speedup in the highly skewed workload. To better understand the decision making in our policy, we will dive deeper into the memory statistics in Section 7.3.3.

7.3.3 *Memory Statistics.*

In this experiment, we show the memory statistics for each data placement policy. We use 4 GPUs and $SF = 160$ on GPU3.8 instance and show the memory statistics for both uniform and skewed query access patterns.e Figure 7 shows the result of our experiment. The y-axis in the Figure indicates the percentage of data (from the entire dataset) that are (1) cached, (2) cached and also replicated across all the GPUs, and (3) not cached in the aggregated GPU memory.

For *Replication-Only*, the data is either cached+replicated or not cached. This results in a total of only 25% of the GPU memory being used to cache data. Since most of the data is not cached, a large portion of query execution will be executed on the CPU which results in significant performance overhead.

For *Partitioning-Only*, the data that are being accessed by the queries can be cached but not replicated across the GPUs. For NVLink-connected GPUs, this results in better performance compared to *Replication-Only* but can result in worse performance in PCIe-connected GPUs.

For *Cache-Aware Replication*, there is more variation of data that are being cached, being replicated, and not being cached. On uniform workload, a total of 84% of the data is being cached, and among those 84%, 8% are replicated across all the GPUs. Conversely, 16% of the data is not cached in the GPUs. Further investigation reveals our policy decides to replicate the smaller dimension tables at the cost of not caching less relevant columns. For example, segments from columns that are used in filters (lo_discount) and aggregation (lo_extendedprice) are not cached since those operations are not expensive on the CPUs (relatively to columns that participate in joins). On skewed workload, our policy decides to cache less data

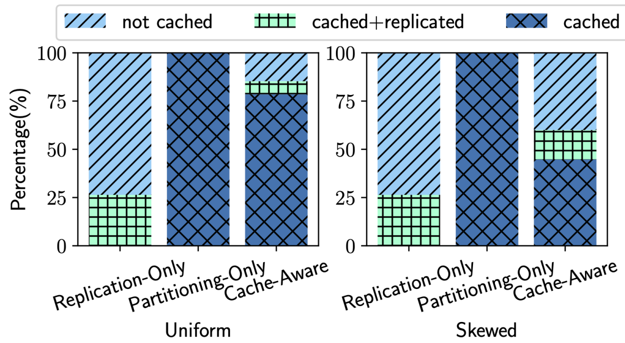


Figure 7: Memory Statistics of Various Data Placement Policies (SF = 160, 4 GPUs)

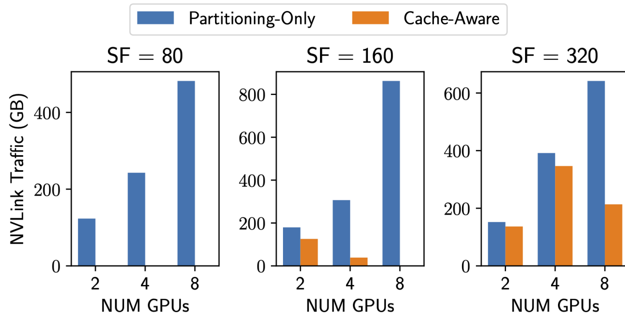


Figure 8: NVLink Traffic of Various Data Placement Policies

in total (60%) but replicate more data at the same time (17%). This is because, for skewed workload, the cold data (data from earlier years) are less relevant since it will be pruned by *min-max filtering*. Instead, the policy will prioritize replicating segments from the dimension tables to reduce the data transfer across GPUs.

7.3.4 *NVLink Traffic.*

To gain deeper insight on each data placement policy, we compare the data traffic going through the NVLink interconnects for each policy. To obtain this metric, we simply aggregate the total *bytes* going across each NVLink channel in each GPU in both directions. Since *Replication-Only* will not incur any data traffic across GPUs, we do not include it in this experiment. We use the uniform data distribution used in Section 7.3.1 for this experiment.

Figure 8 shows the NVLink traffic when running on GPU3.8 instance. For *SF* = 80, *Cache-Aware Replication* incurs almost no NVLink traffic since it manages to fully replicate the dimension tables. For *SF* = 160, *Cache-Aware Replication* has higher traffic for 2 GPUs compared to 4 GPUs. Further investigation reveals that with 2 GPUs, the policy replicates only 2 dimension tables (supplier and date), while with 4 GPUs, it accommodates replication of 3 dimension tables (supplier, date, and customer) by evicting less-performance-critical segments in the fact table (e.g., filter and aggregation columns). A similar trend is observed for SF=320, which results in less traffic for 8 GPUs compared to 4 GPUs.

Overall, across all experiments, *Cache-Aware Replication* reduces the NVLink traffic compared to *Partitioning-Only*. This results in faster query runtime as shown in Figure 6.

7.4 Breakdown of Optimizations in Lancelot

7.4.1 *Speedup after Each Optimization.*

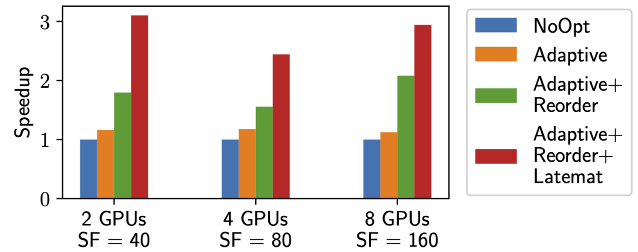


Figure 9: Speedup after Each Optimization in Lancelot

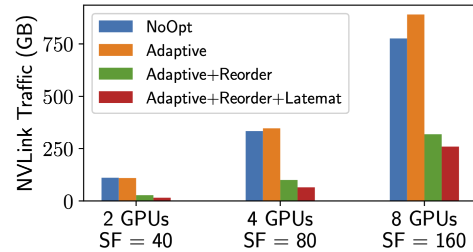


Figure 10: NVLink Traffic after Each Optimization in Lancelot

We now measure the speedup from each optimization applied to Lancelot. We evaluate the performance gain from three optimizations: (1) *late materialization* (Section 5.2.1) (2) *adaptive join strategy* (Section 5.2.2), and (3) *join reordering* (Section 5.2.3). We conduct the experiment on GPU3.8 instance and sweep the number of GPUs alongside the scale factor as shown in Figure 9. To demonstrate the best speedup, we choose a data size in which all the query processing will be done on GPUs. For this experiment, we reduce each GPU cache size from 8GB to 4GB since otherwise, `NoOpt` will suffer from insufficient memory.

When no optimization is applied (`NoOpt`), we default to always use the co-partitioned join as our distributed join strategy. We will join in the order of selectivity starting with the part, supplier, customer, and date table. Without late materialization, we always materialize the relation after each partitioning phase.

Across all scale factors, *adaptive join* (`Adaptive`) gives around 1.2× speedup. With this optimization, broadcast join will be used when joining with smaller tables, whereas co-partitioned join will be used when joining with larger tables.

By incorporating the *join reordering* (`Reorder`), we will reorder the join such that all the broadcast joins will occur first, followed by co-partitioning joins. Doing this can significantly reduce the data transfer between GPUs during co-partitioned joins. We observe up to 2× speedup after applying this optimization.

Finally, *late materialization* (`Latemat`) will give another 1.6× speedup. Our late materialization strategy will reconstruct only the join key after each partitioning phase, and utilize UVA to materialize the rest of the columns lazily during query execution.

7.4.2 *NVLink Traffic.*

To reveal deeper insight, we will show the NVLink traffic after each optimization is applied. Using the same setup as Section 7.4.1, Figure 10 shows the result of this experiment.

After applying the adaptive distributed join strategy (`Adaptive`), we observe higher NVLink traffic for 4 and 8 GPUs. This is because switching from co-partitioned join to broadcast join can potentially increase the amount of data transferred across the GPUs. However,

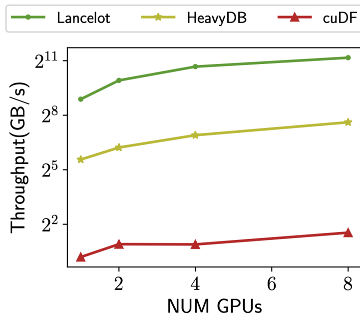


Figure 11: Scalability Comparisons

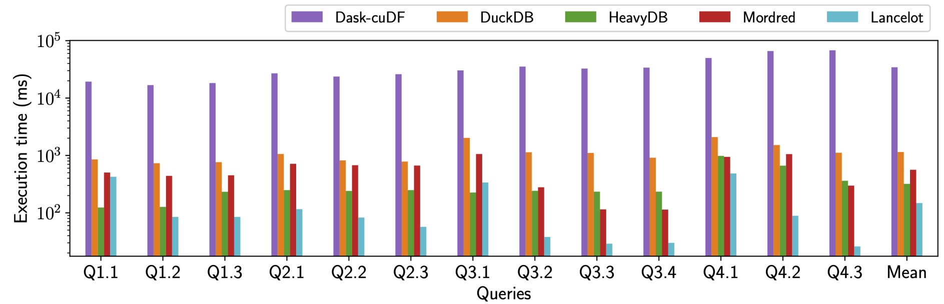


Figure 12: SSB Performance of Different GPU DBMS

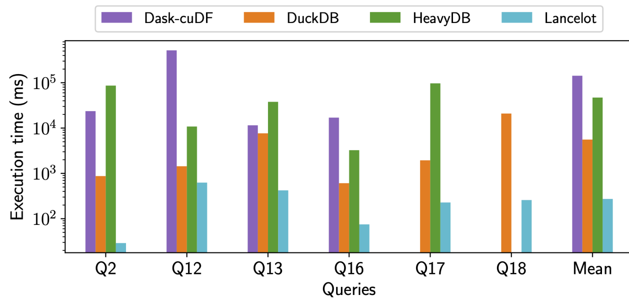


Figure 13: TPC-H Performance of Different GPU DBMS

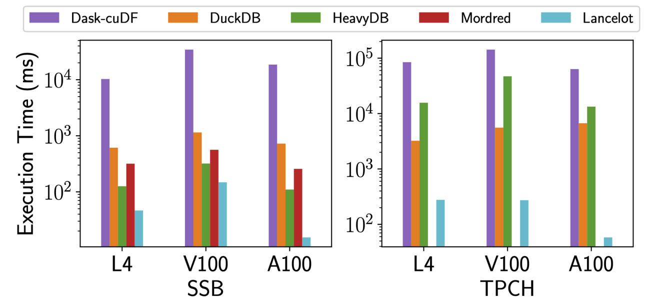


Figure 14: Performance on Different Compute Instances

this does not necessarily translate to slower query execution time since the partitioning phase in co-partitioned join can also introduce non-trivial overhead and is not reflected by the NVLink traffic.

Reordering joins (`Reorder`) significantly reduces the NVLink traffic by up to 2.5×. Moreover, the late materialization (`Latemat`) can further reduce the NVLink traffic by up to 20%. These traffic reductions will translate to the speedup we observe in Section 7.4.1.

7.5 Comparison with Other GPU DBMSes

This subsection reports the end-to-end performance evaluation of four existing CPU/GPU DBMSes:

- **DuckDB:** DuckDB [45] is an embedded CPU-based analytical database which supports columnar engine and parallel execution.
- **Dask-cuDF:** Dask-cuDF [5] is a multi-GPU extension of cuDF [4], a GPU-accelerated DataFrame library. Dask-cuDF caches the data in GPUs using the LRU policy.
- **HeavyDB:** HeavyDB [6] is a commercial multi-GPU DBMS. HeavyDB caches the data in GPUs using the LRU policy and simply partition the data across all the GPUs.
- **Mordred:** Mordred [60] is a hybrid CPU-GPU DBMS which utilizes the *semantic-aware caching* and the *segment-level query execution*. Mordred, however, only supports a single GPU.
- **Lancelot:** Lancelot is our prototype of Hybrid CPU and Multi-GPU DBMS explained in Section 6.

In this Section, we will run four sets of experiments: (1) scalability comparison (Section 7.5.1), (2) query performance on SSB (Section 7.5.2), (3) query performance on TPC-H (Section 7.5.2), and (4) query performance on various hardware configurations (Section 7.5.3). Throughout the experiment, for `Lancelot` and `Mordred`, we allocate half of the GPU memory as the cache size for each GPU. For `HeavyDB` and `Dask-cuDF` we let the system control the GPU memory. For this experiment, instead of scaling up the dimension table size as described in Section 7.1, we conducted the experiment using the original dimension table sizes in SSB and TPC-H.

7.5.1 Scalability Comparison.

Figure 11 shows the scalability comparison between Lancelot and existing GPU DBMSes when running SSB on GPU3.8 instance (V100 GPUs). To avoid GPU underutilization, we conducted a weak scaling experiment — we increase the number of GPUs as the data size increases. Hence, we will use 1 GPU for $SF = 40$, 2 GPUs for $SF = 80$, 4 GPUs for $SF = 160$, and 8 GPUs for $SF = 320$.

The result shows that Lancelot outperforms both HeavyDB and Dask-cuDF in terms of throughput and scalability. The performance difference between Lancelot and other GPU DBMSes boils down to several factors: (1) Lancelot is built on top of Crystal library which has shown to perform better compared to other GPU DBMSes [22]. (2) Lancelot adopts an intelligent data placement strategy discussed in Section 4 which is not employed by other GPU DBMSes. (3) Lancelot adopts various GPU database optimizations listed in Section 5.2 which are not adopted by the existing GPU DBMSes.

Figure 11 also shows that Lancelot throughput is not fully linear when increasing the number of GPUs. This is because as the data size increases, the hash table sizes also increase, reducing join throughput due to higher random access overhead. Since SSB queries involve multiple joins, this will have a significant impact on the query performance.

7.5.2 End-to-end Performance Comparison on SSB and TPC-H.

Figure 12 and Figure 13 shows the query performance when running SSB and TPC-H respectively using 4 V100 GPUs on GPU3.8 instance. In this experiment, we use SF=320 which translates to about 50% of the total input data cached in the GPU memory.

Compared to `DuckDB`, `Lancelot` is 8× faster on SSB and 20× faster on TPC-H. This is due to the performance gap between CPUs and GPUs. Compared to `Mordred`, `Lancelot` is around 4× faster on SSB. This is because `Lancelot` can provide a larger cache size

compared to Mordred by utilizing 4 GPUs. We do not compare to Mordred on TPC-H since it does not support TPC-H queries.

Compared to Dask-cuDF, Lancelot is 232× faster on SSB and 522× faster on TPC-H. Dask-cuDF suffers from out of memory execution which forces the intermediate result to spill to the CPU memory. This will result in back and forth data transfer between CPU and GPUs which degrades performance. On TPC-H, Q17 and Q18 failed since these engines cannot allocate enough memory space for the intermediate results.

Compared to HeavyDB, Lancelot is around 2× faster on SSB. HeavyDB is faster for Q1.1 but significantly slower for the other queries. This is because Lancelot decides to cache columns used in the other query sets (Q2.x, Q3.x, and Q4.x) since Q1.x are mostly just scan queries, which are not expensive when executed on the CPUs. On TPC-H, Lancelot is around 172× faster than HeavyDB. HeavyDB performance suffers heavily due to large intermediate results on complex TPC-H queries, resulting in excessive memory spilling and data transfer between CPUs and GPUs. HeavyDB also failed to execute Q18 due to out of memory execution.

7.5.3 End-to-end Performance on Various Hardware Platforms.

Figure 14 show the query performance when running both SSB and TPC-H on three different instances: GPU3.8 (V100), GPU4.8 (A100), and g6.48xlarge (L4). Across all the instances, we use 4 GPUs and ran the experiment with $SF = 320$. This results in approximately 50% of the input data being cached in the V100 GPUs' memory, 75% in the L4 GPUs' memory, and 100% in the A100 GPUs' memory. The results for GPU3.8 instance are discussed in Section 7.5.2.

Running on the g6.48xlarge instance, we see performance improvement across all systems compared to GPU3.8 instance. Even though L4 has lower memory bandwidth than V100 (330 GB/s vs 1TB/s), the larger GPU memory capacity (24 GB vs 16 GB) and significantly more CPU cores (192 cores vs 108 cores) made up for the difference. Despite slightly larger GPU memory capacity, query failures still occur for HeavyDB (Q18) and Dask-cuDF (Q17 and Q18). On this instance, Lancelot outperforms the the existing systems by at least 3× on SSB and 12× on TPC-H.

Running on the GPU4.8 instance, we observe performance improvement across all GPU DBMSes compared to g6.48xlarge and GPU3.8 instances since A100 has the largest GPU memory capacity (40 GB) and the highest memory bandwidth (1.5TB/s). Nevertheless, query failures still occur on Q18 for HeavyDB. DuckDB runs slower on this instance compared to g6.48xlarge instance due to fewer CPU cores (128 cores vs 192 cores), Overall, Lancelot still outperforms the existing systems by at least 7× on SSB and 115× on TPC-H.

8 RELATED WORK

There have been several previous systems that attempted to leverage multiple GPUs for query execution.

A subset of existing works focused on accelerating individual database operators with multiple GPUs. Paul et al. [44] and Gao et al. [27] introduced a hash join implementation on multiple GPUs. Rui et al. [46] proposed a nested loop, sort-merge and hybrid joins implementation for multi-GPU. Maltenberger and Ilic et al. [53] showcased the advantage of multi-GPU sorting with fast interconnects. All these operator implementations are orthogonal to our work and can be incorporated to Lancelot.

There have also been commercial systems which support multiple GPUs. HeavyDB is an open-source, commercial GPU-accelerated DBMS. HeavyDB caches the most recently used data and partitions it across multiple GPUs. HeavyDB does not have a heterogeneous query execution strategy. When executing data larger than aggregated GPU memory, it will either (1) fallback to query execution on CPUs or (2) execute the query in multiple stages on GPUs.

PG-Storm is a GPU-accelerated PostgreSQL extension. Its core feature is GPUDirect-SQL, which enables reading data directly from NVMe to the GPUs. We do not compare against PG-Storm since the multi-GPU feature is not open-sourced. PG-Storm does not have data placement or hybrid query execution strategy.

cuDF is a GPU-accelerated DataFrame Library from NVIDIA. cuDF can support multiple-GPUs by leveraging Dask [5], hence the name Dask-cuDF. Similar to PG-Storm, Dask-cuDF does not have data placement and a hybrid query execution strategy.

BlazingSQL [2] is a GPU accelerated SQL engine built on top of the RAPIDS [14] ecosystem. Both cuDF and BlazingSQL[2] use the same internal execution engine but provide different API (pandas vs SQL). We compare against Dask-cuDF since it remains actively developed, unlike BlazingSQL, which has been inactive since 2021.

HetExchange [23, 24] is a query execution framework which encapsulates heterogeneous parallelism in CPUs and GPUs through redesigning the classical *Exchange* operator. It supports just-in-time code generation and hybrid CPU and multi-GPU query execution. HetExchange, however, does not address the data placement between CPU and multi-GPU systems and lacks an optimizer component to generate the physical query plan based on the data location.

Finally, HERO [36] proposed an operator placement strategy for CPU and multi-GPU systems that can be completely independent on cardinality estimation of the intermediate result. However, unlike Lancelot, this work only focuses on operator placement and does not address the challenges in other aspects of database design such as data placement and query execution.

9 CONCLUSION

This paper advances the state-of-the-art of GPU DBMS by showing how to leverage both (1) hybrid CPU-GPU, and (2) multiple GPUs DBMS at the same time. To simplify the design space, we introduce the *unified multi-GPU abstraction* which will treat multi-GPU as a single large GPU. We then optimize hybrid CPU and multi-GPU DBMS in two aspects: (1) data placement and (2) query execution. For data placement, we introduce the *cache-aware replication policy* that takes into account the cost of shuffle when replicating data and could coordinate both caching and replication decisions for best performance. For query execution, we extend the existing CPU-GPU query execution strategy with distributed query processing techniques to support multiple GPUs. We then integrate both solutions in Lancelot, our hybrid CPU and multi-GPU DBMS. Our evaluation shows that our *cache-aware replication policy* outperforms other policies by up to 2.5× and Lancelot outperforms existing GPU DBMSes by at least 2× on SSB and 12× on TPC-H.

ACKNOWLEDGMENTS

This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle External Research Office.

REFERENCES

[1] 2024. AMD Instinct MI300 Series Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300.html.
[2] 2024. BlazingSQL. https://blazingsql.com.
[3] 2024. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[4] 2024. cuDF- Performance Comparison. https://github.com/rapidsai/cudf/blob/branch-23.04/docs/cudf/source/user_guide/performance_comparisons.ipynb.
[5] 2024. Dask-CUDA. https://docs.rapids.ai/api/dask-cuda/nightly/.
[6] 2024. HeavyAI. https://www.heavy.ai/.
[7] 2024. HIP Programming Guide. https://github.com/ROCm-Developer-Tools/HIP.
[8] 2024. Kinetica. https://kinetica.com/.
[9] 2024. NVIDIA Collective Communication Library. https://developer.nvidia.com/nccl.
[10] 2024. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/.
[11] 2024. NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/.
[12] 2024. Opencl. https://www.khronos.org/opencl/.
[13] 2024. PG-Storm. https://github.com/heterodb/pg-strom.
[14] 2024. RAPIDS. https://rapids.ai.
[15] 2024. The RAPIDS Accelerator for Apache Spark. https://nvidia.github.io/spark-rapids/.
[16] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (2014), 199–209.
[17] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient co-processor utilization in database query processing. *Inf. Syst.* 38 (2013), 1084–1096.
[18] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-Accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936
[19] Sebastian Breß and Gunter Saake. 2013. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1398–1403. https://doi.org/10.14778/2536274.2536325
[20] Sebastian Breß, Norbert Siegmund, Max Heimel, Michael Saecker, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake. 2014. Load-aware inter-co-processor parallelism in database query processing. *Data Knowl. Eng.* 93 (2014), 60–79.
[21] Sebastian Breß, Felix Beier, Hannes Rauhe, Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. 2012. Automatic Selection of Processing Units for Coprocessing in Databases, Vol. 7503. 57–70. https://doi.org/10.1007/978-3-642-33074-2_5
[22] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (nov 2023), 441–454. https://doi.org/10.14778/3632093.3632107
[23] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760
[24] 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. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf
[25] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734
[26] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.* 13, 6 (mar 2020), 884–897. https://doi.org/10.14778/3380750.3380758
[27] Hao Gao. 2021. Scaling Joins to a Thousand GPUs. In *ADMS@VLDB*. https://api.semanticscholar.org/CorpusID:237250537
[28] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886
[29] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (dec 2009), 39 pages. https://doi.org/10.1145/1620585.1620588
[30] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *SIGMOD*.
[31] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833
[32] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB* (2013).
[33] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-Cache Query Co-Processing on Coupled CPU-GPU Architectures. *Proc. VLDB Endow.* 8, 4 (dec 2014), 329–340. https://doi.org/10.14778/2735496.2735497
[34] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* (2013).
[35] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *DaMoN*.
[36] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (mar 2017), 733–744. https://doi.org/10.14778/3067421.3067423
[37] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.
[38] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705
[39] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3514221.3517911
[40] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1951–1960. https://doi.org/10.1145/2882903.2903735
[41] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. (2023). https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf
[42] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.
[43] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-Time Compilation Based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (nov 2020), 202–214. https://doi.org/10.14778/3425879.3425890
[44] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. *MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254
[45] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212
[46] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927
[47] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, 17.
[48] Anil Shanbhag, Bobbi Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 ACM SIGMOD international conference on Management of data*.
[49] Anil Shanbhag, Xiangyao Yu, and Samuel Madden. 2020. A Study of the Fundamental Performance Charecteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data*. ACM.
[50] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *Hardware-conscious Hash-Joins on GPUs*. Technical Report.
[51] Evangelia A Sitaridi and Kenneth A Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. ACM, 4.
[52] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. ACM.

[53] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM Manag. Data* 1, 1, Article 29 (may 2023), 26 pages. https://doi.org/10.1145/3588709

[54] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[55] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. 1982. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.* 7, 3 (sep 1982), 323–342. https://doi.org/10.1145/319732.319734

[56] H. Vandierendonck and P. Trancoso. 2006. Building and Validating a Reduced TPC-H Benchmark. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*. 383–392. https://doi.org/10.1109/MASCOTS.2006.16

[57] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 107–118. https://doi.org/10.1109/MICRO.2012.19

[58] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on GPUs: A closer look. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2663–2673.

[59] Bobbi Yogatama, Brandon Miller, Yunsong Wang, Graham Markall, Jacob Hemstad, Gregory Kimball, and Xiangyao Yu. 2023. Accelerating User-Defined Aggregate Functions (UDAF) with Block-wide Execution and JIT Compilation on GPUs. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 19–26. https://doi.org/10.1145/3592980.3595307

[60] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2491–2503.

[61] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB* (2013).

[62] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. 2015. Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources. *Journal of Computer Science and Technology* 30 (2015).

[63] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. 2013. OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1374–1377. https://doi.org/10.14778/2536274.2536319