Scheduling for HTAP systems on CPU-GPU clusters

Krati Agrawal   Arjun Balasubramanian   Shivangi Kamat   Giri Prasanna Mugunda Krishnan

University of Wisconsin - Madison

Abstract

HTAP systems that run a combination of OLAP and OLTP queries can be built to leverage the advantages offered by the next generation hardware such as GPUs and accelerators. GPUs have abundant parallelism and high memory bandwidth, and thus there has been considerable interest in utilizing GPUs to accelerate OLAP workloads. In this paper, we explore the idea of how a cluster of CPU-GPU co-processor servers can be used to accelerate HTAP workloads. We tackle the interesting problem of scheduling this mixture of queries across a heterogeous cluster by focusing on efficient query routing, and GPU memory management. We propose EEVEE [1], a heuristic-based scheduler for HTAP systems that performs intelligent scheduling decisions to improve overall latency of query execution. We implemented a simulator to evaluate the performance of executing HTAP queries on a cluster of CPUs and GPUs. We evaluate our design on a series of micro-benchmarks and SSB workloads and obtain gains of up to $6X$ by using a heterogeous cluster, and our heuristic scheduling policy results in $4X$ improvement in the makespan compared to our baseline policy.

1 Introduction

Database systems have become commonplace and are widely adopted at large scale by multiple organizations [28, 39]. Traditionally, database systems are designed to serve one of the following two purposes -
(i) Online Transaction Processing (OLTP). OLTP systems are designed to handle a high volume of concurrent transactions. These transactions typically touch only a small fraction of the resident data and can therefore be executed quite fast. According to the standardized TPC-C benchmark [2], today's state-of-the-art OLTP systems can execute more than a million transactions per second. Examples of OLTP systems include banking transaction processing and online sales processing.
(ii) Online Analytical Processing (OLAP). About two decades ago, a new use-case for the usage of database systems emerged - Large companies wanted to use database systems for running Business Intelligence (BI) queries to obtain insights for data-driven decisions. These OLAP/BI queries were different from OLTP queries - The queries were typically long-running and involved scanning through a large portion of the resident data. Examples of OLAP queries include aggregated online sales statistics by seller and geographical region.

Initial attempts were made to run these OLAP queries on OLTP systems. However, this resulted in a number of critical issues. Since OLAP queries are long running and read a large amount of data, it severely restricts the ability of OLTP queries to lock and update data. This in turn leads to *poor concurrency* and head-of-line blocking for OLTP queries. Additionally, since OLAP queries are CPU intensive since they typically involve a number of join operations. This leads to severe *resource contention* and thereby affects the running time of latency sensitive OLTP transactions.

The data staging architecture [19] was devised to overcome the above mentioned problems. The idea was to carry out transaction processing in a dedicated OLTP system. Separate *data warehouses* were installed to serve OLAP queries. Periodically, data changes from the OLTP system are extracted and loaded into the data warehouse through a process known as **Extract-Load-Transform (ETL)**.

The delegation of OLTP and OLAP query processing to separate systems presented a number of advantages. It allowed OLTP queries to run at high concurrency and not suffer from contention from OLAP queries. It also paved the way for independent optimizations in these systems such as the use of columnar storage format for OLAP systems [3, 36]. However this architecture traded off the advantages with one key disadvantage - the ETL pipeline is executed only periodically and this means that OLAP queries more often than not operate on a *stale* copy of the data.

Recently, there has been a strong urge to design database

[1]Eevee is a Pokémon that evolves into eight different pokemon through various methods. E.g. It evolves into Flareon when exposed to a Fire Stone, or Vaporeon when exposed to a Water Stone, and so on. In our work, EEVEE performs intelligent query routing onto CPUs or GPUs depending on the type of query.

systems that can support *real-time* business intelligence [5]. The current ETL pipeline architecture is fundamentally at odds with this new requirement. Consequently, database designers have advocated building **Hybrid Transactions and Analytics Processing (HTAP)** systems that can support both OLTP and OLAP queries in a single system. The idea here is to not move analytical processing in its entirety to HTAP systems, but only an important subset of queries that can benefit from real-time data. Thus, the design of HTAP systems has been a hot research topic amongst researchers from academia and industry [13, 21, 29, 34].

In another line of work, researchers have been looking at how to speed up query processing by using emerging technology such as smart storage devices [12, 22, 33, 37], custom accelerators [42], and low latency networks [6, 9, 24, 45]. Thus, it would be interesting to look at how HTAP systems can be designed to leverage the benefits of these technologies.

Thus, we propose an initial design of EEVEE, a HTAP system that can handle a mixture of OLTP and OLAP queries and schedule them across a heterogeneous cluster comprising of a CPU and multiple GPUs. EEVEE is designed to be an *in-memory, deterministic* database, where all tuples can fit in CPU DRAM. GPU memory on the other hand has limited capacity and can hold only a subset of the tuples. EEVEE divides the incoming queries into epochs and uses a novel scheduling algorithm to decide where each transaction must execute. For sake of simplicity, EEVEE assumes that OLTP queries can execute only on CPUs while OLAP queries can execute on either a CPU or a GPU. EEVEE uses MVCC [8] for concurrency control and data versioning and offers strict linearizability guarantees. Finally, EEVEE consists of a memory manager per GPU that implements a novel *two-level* eviction policy to manage limited GPU DRAM.

In this work, we focus on two important and novel aspects related to our vision of EEVEE -

Scheduling - Scheduling plays an important role in ensuring that shared clusters operate at high utilization. Resource management in this regard is a challenging problem and has been explored in diverse domains such as CPU scheduling [10, 23, 25], congestion control in networks [4, 17, 27, 41], big data task scheduling [11, 14, 15, 18, 38, 40, 44], and GPU cluster scheduling [16, 26, 31, 32, 43]. For EEVEE, we require a scheduler that accounts for the *heterogeneity* of the underlying hardware substrates in the cluster and also be aware of the *overheads* incurred by transferring data over interconnects that connect the hardware devices. No previous scheduler accounts for these factors and hence we propose a new scheduling algorithm for HTAP scheduling. The scheduling algorithm needs to make two decisions. First, the algorithm must decide the best location (CPU/GPU) to execute each transaction. Second, the algorithm must finalize an order for the execution of the transactions.

GPU Memory Management - GPUs have limited DRAM capacity which needs to be managed properly. The goal of a GPU memory manager should be to keep tuples in memory that can potentially benefit future transactions that execute on the GPU. This would help reduce the interconnect overheads associated with offloading the query execution to the GPU. EEVEE adopts a memory management policy that imposes *lazy eviction*, i.e, tuples are evicted from GPU memory only when there is no memory available to accommodate new incoming tuples. We propose a two-level eviction algorithm to decide which tuple(s) to evict in such a scenario.

To study scheduling and GPU memory management, we built a discrete event-based simulator in Java that enables us to study the regimes imposed by our scheduling algorithm and compare GPU memory management policies. The simulator is completely configurable and is built as a collection of pluggable modules. We have open-sourced our simulator at https://github.com/Arjunbala/HTAP-Scheduling to enable researchers to further study and quickly evaluate various aspects of HTAP systems.

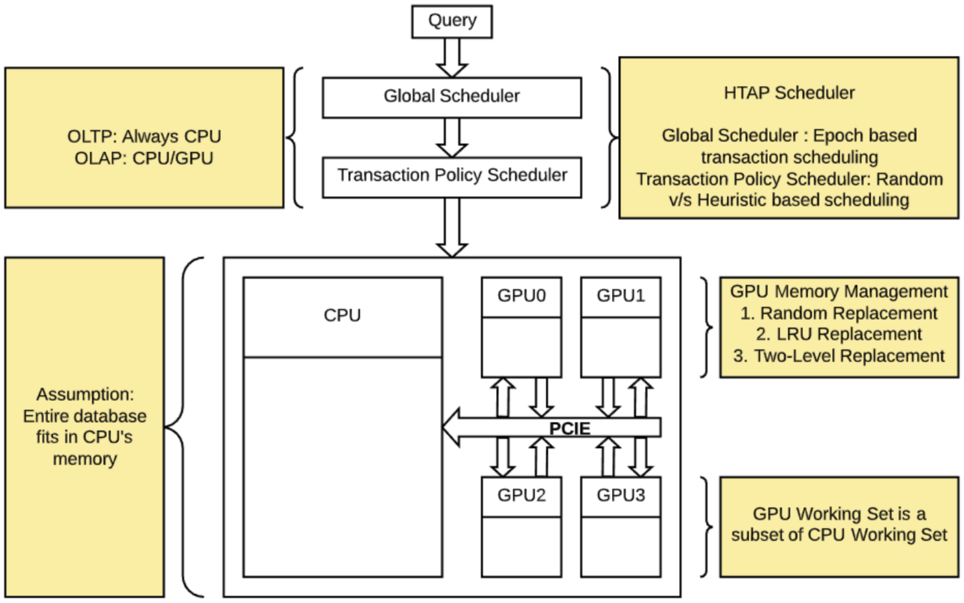
We evaluate the proposed scheduling algorithm and two-level eviction policy on the simulator using the StarSchema Benchmark (SSB) [30] with a scale-factor of 1. Our results show that the scheduling algorithm and two-level eviction can offer upto ~6X improvement in makespan compared to baseline approaches. Additionally, we present a set of microbenchmarks that evaluate the behavior of the algorithms under a variety of conditions. Finally, we perform sensitivity analysis to illustrate the regimes imposed by our scheduling algorithm.

The rest of this paper is arranged as follows. Section 2 covers our HTAP system architecture and explains the scheduling policy and GPU memory management in detail. Section 3 constructs the implementation of EEVEE simulator and explains its components. Section 4 then evaluates EEVEE system with workloads and discusses the observed trends. Section 5 proposes some avenues for further extension of this work, and finally section 6 concludes this paper.

2 HTAP System Architecture

EEVEE is an in-memory CPU database scheduler, which efficiently schedules OLTP and OLAP transactions on a hybrid hardware consisting of one CPU and multiple GPUs. We make two assumptions - first, the CPU memory is big enough to house the entire database, and second, that OLTP transactions can only be scheduled on a CPU, while OLAP transactions can be scheduled on either hardware. An incoming query first goes through a Global Scheduler, which divides transactions into epochs. The epoch time, which is how long an epoch lasts, is a configurable parameter. Transactions in each epoch are then batched and sent to the Transaction Policy Scheduler.

Transactions need to be allotted a device for execution, i.e, CPU or one of the GPUs, and also ordered for execution on the device at which they are scheduled. These critical decisions



Query
Global Scheduler
Transaction Policy Scheduler
OLTP: Always CPU
OLAP: CPU/GPU
HTAP Scheduler
Global Scheduler : Epoch based transaction scheduling
Transaction Policy Scheduler: Random v/s Heuristic based scheduling
CPU
GPU0
GPU1
GPU2
GPU3
PCIE
Assumption: Entire database fits in CPU's memory
GPU Memory Management
1. Random Replacement
2. LRU Replacement
3. Two-Level Replacement
GPU Working Set is a subset of CPU Working Set

Figure 1: EEVEE *HTAP System*

are taken by the Transaction Policy Scheduler, which performs this decision making at the granularity of each epoch for a batch of transactions in that epoch. Parameters such as speed ups offered by GPUs for OLAP queries, and the PCIe overheads factor into this decision making process (Section 2.1).

The database maintains the version count for each tuple so as to provide concurrency control between transactions. When OLTP transactions are run on a CPU which updates a tuple data, the version count for that tuple is incremented. This is similar in spirit to MVCC [8]. While older versions of data can exist in the GPU memory, the tuples will be updated before any new OLAP transactions asks for that tuple to guarantee linearizability.

In our system, the CPU memory is large enough to hold the database in its entirety. However, the GPU memory can only hold a subset of tuples. We assume that at any moment, GPU memory is large enough to at least store the tuples required to execute the current OLAP query on that GPU. EEVEE has a memory management module for maintaining the tuples in GPU memory. This is further discussed in section 2.2 where we elaborate on the various eviction policies employed by EEVEE.

To preserve the data integrity of conflicting transactions, EEVEE guarantees linearizability in conflicting transactions. Tuples can be read by multiple transactions in parallel, but only one transaction can update a tuple at a time. Hence, tuples which are to be updated are protected by locks. For resolving conflicts, each transaction grabbing a write tuple, also updates the time when the transaction will complete and hence release the lock. When a second transaction tries to grab the lock for the same tuple, it is not granted the lock and is made to back off until the previous transaction completes.

2.1 HTAP Scheduling Policy

In EEVEE, a scheduler accepts a list of queries, each of which may be an OLAP or OLTP query. A scheduler must decide on where each query must execute - this may either be the CPU or one of the GPUs in the cluster. Additionally, a scheduler must decide on the order in which transactions execute while providing the serializibility guarantee.

First, we establish a set of assumptions that we make regarding hardware capabilities. For a CPU with N cores, we assume that a CPU can execute atmost $\frac{N}{4}$ transactions in parallel at any given point of time. We further assume that a GPU can only execute one transaction at a time. We make this assumption since current abstractions for spatially multiplexing the GPU such as CUDA streams [1] do not offer perfect isolation guarantees.

We now describe the list of factors that we consider important while deciding which transaction should get more priority for executing on a particular GPU. First, we believe that the relative speedup obtained by running a particular transaction on a particular GPU is an important factor that needs to be considered. Due to the nature of its operations, certain kinds of OLAP queries may benefit more from GPU acceleration than others [35]. Additionally, differences in speedup may arise due to difference in GPU generations (e.g. Volta vs Pascal). Second, we believe that it is more favorable to schedule those transactions that incur lesser PCIe overheads due to data movement over the interconnection network. Several factors influence the amount of data that needs to be transferred. It is preferable to schedule those OLAP queries that have a smaller

Pseudocode 1 EEVEE Scheduling Algorithm

```
 1: TRANSACTION                                   ▷ Each individual transaction
 2: TRANSACTIONLIST                                   ▷ List of transactions
 3: $\alpha$ [0,1]                        ▷ Knob for transaction scoring mechanism
 4: $\beta$ [0,1]                        ▷ Knob for transaction assignment to GPU
 5:
 6: ▷ Given a list of transactions, decide the placement and execution order of these
    transactions
 7: procedure SCHEDULETRANSACTIONS(TRANSACTIONLIST TL)
 8:     TRANSACTIONLIST conflictingOLTP = GETOLTPCONFLICTS(TL)
 9:     Schedule transactions in conflictingOLTP on CPU ordered by accept stamp
10:     TRANSACTIONLIST remainingTransactions = TL - conflictingOLTP
11:     TRANSACTIONSCORES = { }
12:     for TRANSACTION T in remainingTransactions do
13:         if T is OLAP then
14:             for GPU id in the cluster do
15:                 TRANSACTIONSCORES[T][gpuID] = COMPUTESCORE(T, id)
16:             end for
17:         else
18:             Consider T for execution on CPU, ordered by timestamp
19:         end if
20:     end for
21:     while any OLAP transaction remains unscheduled do
22:         TRANSACTION T, GPU g1 = MAXSCORE(TRANSACTIONSCORES)
23:         GPU g2 = GETLEASTASSIGNEDGPU()
24:         Generate random number R between 0 and 1
25:         if R $\leq \beta$ then
26:             Schedule T on GPU g1
27:         else
28:             Schedule T on GPU g2
29:         end if
30:         Remove T from consideration while finding max score
31:     end while
32: end procedure
33:
34: ▷ Compute alignment score of a transaction towards a particular GPU
35: procedure COMPUTESCORE(TRANSACTION T, GPU gpu)
36:     return $alpha * \frac{t_{cpu}}{t_{gpu}}$ + (1-$alpha$)*(1-PCIEOVERHEADS)
37: end procedure
38:
39: ▷ Get the highest remaining score
40: procedure MAXSCORE(TRANSACTIONSCORES scores)
41: end procedure
42:
43: ▷ Get the GPU with least assigned transaction time
44: procedure GETLEASTASSIGNEDGPU
45: end procedure
46:
47: ▷ Get the OLTP transactions that conflict with OLAP queries
48: procedure GETOLTPCONFLICTS(TRANSACTIONLIST TL)
49: end procedure
```

read set on a GPU since it minimizes the amount of data that needs to be transferred from the host to the device. On similar lines, it is preferable to schedule those transactions on a GPU for which tuples from the read set already reside on the GPU. Additionally, we must consider the size of the query output since this will determine the device to host transfer time.

To model the above factors, we compute the below quantities for every OLAP queries t on every GPU -

$$Speedup_{t,gpu} = t_{cpu}/t_{gpu} \tag{1}$$

where t_{cpu} is the running time of the transaction on CPU, while t_{gpu} is the running time of the transaction on the particular GPU.

$$Overheads_{t,gpu} = HToD(t,gpu) + DToH(t) \tag{2}$$

where $HtoD$ is the time required to transfer required tuples from the read set from the CPU to the GPU. Note that this excludes those tuples that are already resident on the GPU. $DToH$ is the time required to transfer the query output from the GPU to the CPU.

Now, a query t has greater affinity towards a GPU gpu if $Speedup_{t,gpu}$ is more and $Overheads_{t,gpu}$ is lesser. Thus, for every transaction t and GPU gpu, we assign a score that captures both of these factors as below (Line 35 in Algorithm 1) -

$$Score_{t,gpu} = \alpha * Speedup_{t,gpu} + (1-\alpha) * (1 - Overheads_{t,gpu}) \tag{3}$$

where α is a knob used to adjust the relative importance of the two factors. We perform sensitivity analysis over α in Section 4.3.

The final algorithm for scheduling transactions is presented in Algorithm 1. As a first step, we identify those OLTP transactions that conflict with any OLAP query. We then schedule such transactions on the CPU with immediate effect. This is to ensure that such OLTP transactions do not end up getting queued behind long-running OLAP queries. Next, for each OLAP query and every GPU pair, we compute a score as described in equation 3.

Now, while any OLAP query remains to be scheduled, we greedily pick the query and GPU pair that has the maximum score. Once we pick such a query, we can either schedule the query on the GPU that was associated with the maximum score (we term this as *best fit* GPU) or on a GPU that has the least amount of OLAP query execution time currently assigned it (we term this as *least loaded* GPU). We use a knob β to decide whether to schedule the query on the best fit or the least loaded GPU. As shown in line 24 of Algorithm 1, we generate a random number between 0 and 1. If the generated number is less than β, we schedule the query on the best fit GPU and otherwise on the least loaded GPU. We perform sensitivity analysis of knob β in Section 4.3.

With the above steps, we have decided the location for execution of each query. We then proceed to execute the query in the order of their accept-stamps. In case of a tie, we schedule the shorter query first for execution.

2.2 GPU Memory Management

While we have an effective HTAP Scheduling policy, we need a robust memory management policy for our system. GPU DRAM is severely limited in terms of capacity (typically few GBs), and significant clock cycles are incurred in transferring useful tuples over the PCIe interconnect. Every tuple on a GPU has an extra field which logs a timestamp of when it was loaded onto the GPU, or when it was last accessed as part of a query. In our system, we assume that the GPU is able to hold all the tuples necessary to execute an OLAP transaction. EEVEE adopts a two-level eviction algorithm for memory management, but also provides capabilities to switch to simpler management modes like Least Recently Used (LRU).

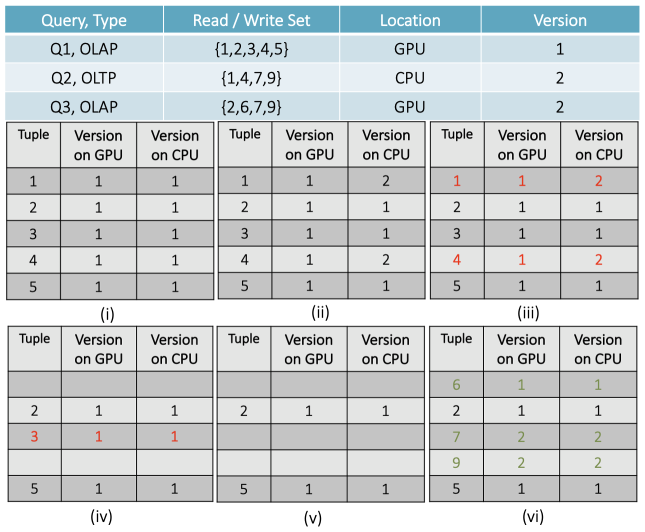
2.2.1 Least Recently Used (LRU) Replacement Policy

Least Recently Used (LRU) is a type of recency-based replacement policy that considers a tuple's latest reference within its lifetime in its decision making. LRU preferentially evicts the least recently used tuples in the GPU's working set. This is accomplished by tuples maintaining a timestamp that log the last access time of the tuple within the GPU. LRU starts replacing tuples with the earliest timestamps to make room for incoming tuples. Our algorithm might suffer from thrashing if future queries executing on the GPU require tuples that were evicted as a result of LRU replacement decisions.

2.2.2 Two-Level Replacement Policy

We propose an intelligent two-level replacement policy that considers the fact that OLTP queries execute exclusively on the CPU, while OLAP queries execute on either the CPU or the GPU. An OLTP query executing on the CPU might change the values of tuples that are currently residing in other GPUs. To keep track of such changes, we maintain a version number (VN) with every tuple in the device working set. A change in a tuple due to an OLTP query only increments its version number in the CPU's copy. When a GPU needs to make space to accommodate incoming tuples, it first compares the version number of all the tuples in its current working set, and flags all the tuples that are no longer up-to-date on the GPU. It removes all these tuples from the working set, as they would anyway require a transfer from the CPU to ensure the query is working on the correct data. This is the first-level of our policy, and we stop here if it satisfies our memory requirement. If we still require more memory on the GPU, we deploy a second-level LRU policy which evicts other tuples in the working set in the same manner described in the previous section until we make enough room for the incoming tuples. As part of this policy, we also perform a regular clean-up of all stale tuples on the GPU due to the first-level eviction.

The two-level eviction policy is explained in figure 2. Consider an execution with three queries, as shown in the table at the top of the figure. First, an OLAP query is executed on GPU, for which the scheduler imports tuples required by that query into GPU memory, as shown in (i). The second query, which is an OLTP query is executed on the CPU, which updates the tuple versions, as shown in (ii). When the third query, which is an OLAP query is scheduled on GPU, the scheduler needs to evict some tuples (precisely 3 in this case) to make room for the tuples required by the incoming query. In the two-level eviction policy, the first level evicts the modified tuples for which the tuple version on GPU is older than that on the CPU. This is highlighted in red in (iii). Further, as more memory is required, the tuples are evicted based on a



Query, Type	Read / Write Set	Location	Version
Q1, OLAP	{1,2,3,4,5}	GPU	1
Q2, OLTP	{1,4,7,9}	CPU	2
Q3, OLAP	{2,6,7,9}	GPU	2

(i)

Tuple	Version on GPU	Version on CPU
1	1	1
2	1	1
3	1	1
4	1	1
5	1	1

(ii)

Tuple	Version on GPU	Version on CPU
1	1	2
2	1	1
3	1	1
4	1	2
5	1	1

(iii)

Tuple	Version on GPU	Version on CPU
1	1	2
2	1	1
3	1	1
4	1	2
5	1	1

(iv)

Tuple	Version on GPU	Version on CPU
2	1	1
3	1	1
5	1	1

(v)

Tuple	Version on GPU	Version on CPU
2	1	1
5	1	1

(vi)

Tuple	Version on GPU	Version on CPU
6	1	1
2	1	1
7	2	2
9	2	2
5	1	1

Figure 2: Two-level eviction policy example. Tables (i) to (vi) show data about tuples in GPU memory (i) Before executing Q1 (ii) After Q2 is executed on CPU, (iii) Two-level eviction policy: level 1 (iv) Two-level eviction policy: level 2 (v) Post eviction of tuples (vi) Tuples imported for Q3

policy (for example LRU or Random) as shown in (iv). This is done until there is enough space for the tuples for the scheduled transaction (iv) and then the new tuples are imported into GPU memory as in (vi).

3 Simulator Implementation

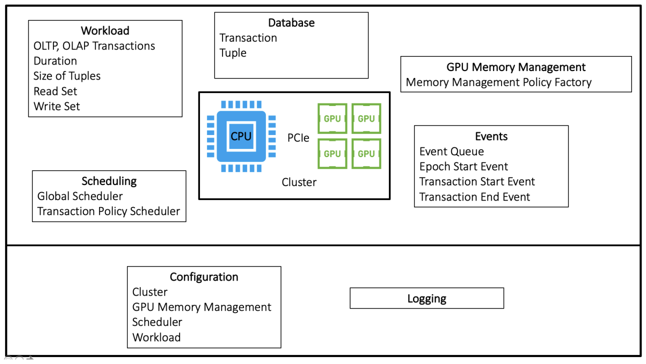


Figure 3: EEVEE *Implementation*

EEVEE is a java based, event-driven, single threaded HTAP simulator. The simulator instantiates a cluster consisting a multi-core CPU and several GPUs, each with a definite memory size. The simulator has been carefully designed with configurability in mind, hence there are several knobs with which the user can parameterize EEVEE to best fit with their application requirements.

The simulator code can be divided into five blocks as shown in the Figure 3. These five components interact with one another to provide for the correct functionality for EEVEE.

Workload defines the queries run on EEVEE and stores the data related to each query. For example the duration of OLTP transactions on CPU and OLAP transactions on both CPU and the GPU. They also hold the read and write sets for each query.

Database describes the structure of transactions and tuples and also holds the associated metadata required for scheduling the transactions. For example, for each transaction, the metadata associated with it would include its ID, time of arrival, the running time estimates on each hardware device, whether it is OLAP or OLAP, and its read and write sets.

Events describes an Event Queue, where transactions are enqueued for scheduling. This global event queue helps maintain ordering between epochs and handles transactions in the queue. Events also hold transaction start and end events, which basically help the simulator to perform activities required for that action. For example, at the start of a transaction, the simulator records the start time, priority of the transaction and the device on which the transaction gets scheduled.

Scheduling covers the heart of EEVEE , which describes the global scheduler and the transaction policy scheduler. The Java class Transaction Ordering Policy Factory integrates the policies implemented into effect. Currently, EEVEE has two ordering policies, heuristic and random. As described in section 2.1, the scheduler also incorporates parameters such as speed ups offered by GPU, PCIe overheads, and scheduling OLAP on a best fit GPU versus a least loaded GPU, which are configurable at the setup time.

The GPU memory management is another critical feature of our scheduler. Currently, EEVEE provides three eviction policies for managing GPU memory, namely LRU, Random and Two-level. To add another policy, one simply needs to add the policy to the Memory Management Policy Factory class and extend the Memory Management class to code in their policy. This seamlessly combines the new policy with the simulator and can be passed in as a configuration while setting up the simulator.

For each component described above, there are several configurable parameters which can be set up before simulating EEVEE .

4 Evaluation and Results

We evaluate EEVEE using our event-driven simulator on a cluster comprised of 32 CPUs and 2 GPUs. We use SSB (StarSchema benchmark) for the evaluations and compare our HTAP scheduler against three other baselines which we describe below. We perform some initial measurements on the workload and feed the data obtained to the simulator which can be approximated as the data obtained from the query optimizer such as [7] before the execution begins. We use a variety of metrics to show that EEVEE performs significantly better than the other baselines on SSB workload.

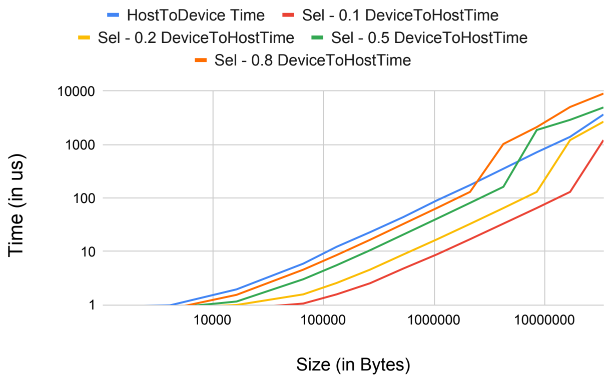


Figure 4: Host to Device and Device to Host transfer time

Workload. For the OLAP queries, we use StarSchema benchmark (SSB) in our evaluation. SSB consists of five tables - part, customer, supplier, date and lineorder. We use a scale factor of 1 to generate the SSB tables.

In order to simulate the OLTP transactions, we used TPC-C benchmark for our experiments. However, since an HTAP system has only one set of tables for OLAP and OLTP transactions, the data extracted from the offline TPC-C experiments is used on SSB tables in the simulator.

Initial measurements. We extract important characteristics from the workloads relevant to our evaluations and feed them into the simulator. We perform offline experiments to calculate - i) the execution times on CPU for OLTP, and both CPU and GPU for OLAP queries, ii) PCIe overheads for transferring data from host to device and vice versa, and iii) building read/write sets of the transactions.

Figure 4 shows the time taken to transfer data from host (CPU) to device (GPU), as well as device to host for different selectivity ratios. These results are used to determine the PCIe overheads based on the amount of data transferred between CPU and GPU during our simulations.

The results from these experiments are assumed to be the data provided by a query optimizer and is therefore known before the query execution begins on the simulator.

Baselines. We compare EEVEE (i.e. heuristic query routing with two-level memory management on GPU) against three baselines: i) CPU-only system, ii) Random query routing with LRU memory management, and iii) heuristic query routing with LRU memory management. These evaluations help us highlight the gains achieved from heuristic and two-level memory management individually.

Metrics. We use a variety of metrics to evaluate EEVEE:
Makespan : Makespan is used to compare the overall run time of a fixed set of transactions on EEVEE and different baselines.
Fairness of resource utilization : We use Jain's fairness index [20] to determine the fairness of resource utilization. This helps evaluate the ability of the scheduler to spread work

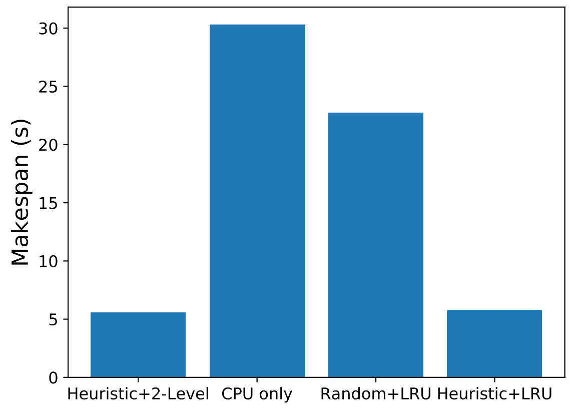


Figure 5: Comparison of makespan against different baselines

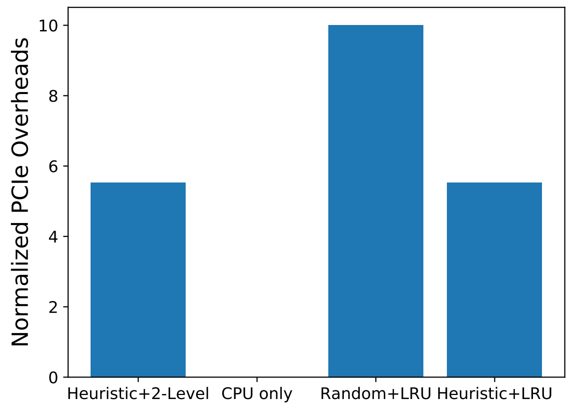


Figure 6: Comparison of PCIe overheads against different baselines

across the cluster and avoid local hotspots. It is computed using the following equation:

$$f(x_1, x_2....x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \quad (4)$$

Here, n is the number of available resources, and x_i denotes the total time for which i^{th} resource is active. The index, f, ranges from $\frac{1}{n}$ (worst case) to 1 (best case).

Normalized PCIe Overheads : We calculate the PCIe overheads per OLAP query for transferring tuples from the main memory to the GPU devices and back before making a scheduling decision.

Latency and Queuing Delay : We present the distribution of times to execute a transaction (latency), as well as the time spent in waiting for its turn to execute (queuing delay) for all transactions.

4.1 Macrobenchmarks

We evaluate EEVEE against the three baselines on SSB workload. The cluster consists of 32 CPU cores and 2 GPUs, and each GPU device has 30MB memory. The epoch time is 200ms. We use $\alpha = 0.5$ and $\beta = 0.0$ which is derived from our sensitivity analysis presented in Section 4.3. The OLAP queries comprise 25% of the total transactions. Requests arrive at a constant rate of 50 requests per second at the scheduler.

Figure 5 shows the makespan of EEVEE against the different baselines. We see 6X improvement over CPU-only system by using a heterogeneous system, and 4X gains over random query routing by optimizing the scheduler to take into account the speedup obtained from GPUs and PCIe overheads for OLAP queries. The gains due to two-level memory management policy over LRU are minor but this could be workload dependent, and we might be able to see better results with more workloads. Figure 6 presents the PCIe overheads. The CPU-only system does not have any GPUs and therefore no PCIe transfers which is indicated by a blank space. EEVEE sees a 2X reduction in PCIe overheads over the random policy because the heuristic policy ensures that the GPU which already has the tuples from the read set is given priority.

Figure 7 and 8 show the distribution of latency and queuing delay respectively for EEVEE and all the baselines. We see that EEVEE decreases the latency and queuing delay by a factor of 4X over random policy and 6X over CPU-only system.

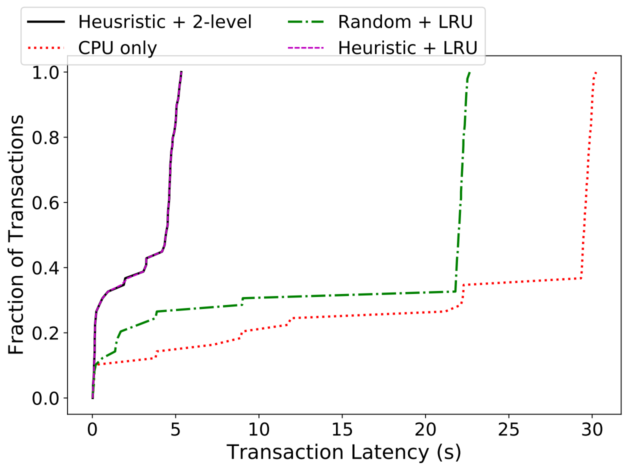


Figure 7: Comparison of latency against different baselines

4.2 Microbenchmarks

4.2.1 Cluster Size

We evaluate the impact on makespan and PCIe overheads as we increase the number of GPUs in the cluster as shown in

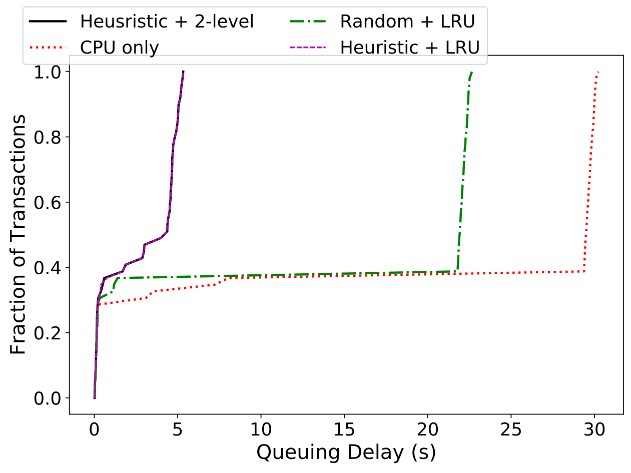
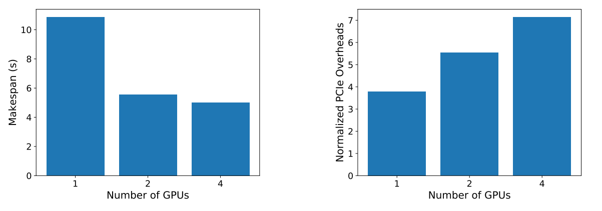


Figure 8: Comparison of queuing delay against different baselines

Figure 9. As expected, the overall execution time reduces with more GPUs as there are more resources available to run the transactions in parallel. The PCIe overheads per OLAP query increase because more tuples need to move across the interconnection network (PCIe) with increase in number of GPUs.

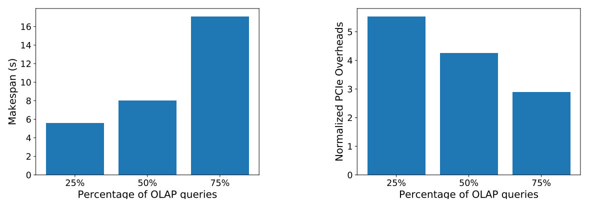


(a) Impact on Makespan (b) Impact on PCIe overheads

Figure 9: Impact of cluster size

4.2.2 Ratio of OLAP vs OLTP transactions

We increase the percentage of OLAP queries from 25% to 50% and 75% and observe the impact on makespan and PCIe overheads. Figure 10 shows that increasing the percentage of OLAP queries increases the makespan since OLAP are longer running transactions than OLTP. The PCIe overheads per OLAP query come down because most of the OLAP queries execute on tuples that are already residing in the GPU memory.

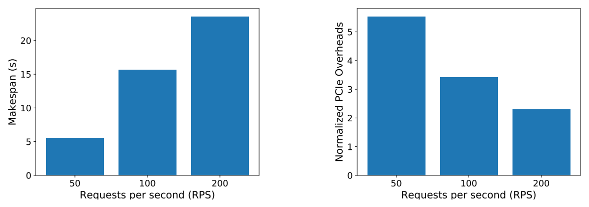


(a) Impact on Makespan (b) Impact on PCIe overheads

Figure 10: Impact of ratio of OLAP vs OLTP

4.2.3 Requests per second

Requests per second (RPS) parameter determines the number of transactions that are received per second at the simulator. An increase in RPS naturally increases the overall execution time of the simulation because of more number of transactions. The PCIe overheads per OLAP query reduce because number of OLAP queries increase and most of them execute on tuples already present in GPU memory, same as what we discussed in Section 4.2.2.



(a) Impact on Makespan (b) Impact on PCIe overheads

Figure 11: Impact of Requests per second

4.3 Sensitivity Analysis

We perform sensitivity analysis over knobs α and β described in Section 2.1 to study the impact on makespan and fairness of resource utilization. Figure 12 shows that lower values of β (more specifically, $\beta = 0.0$) result in shorter makespan indicating that splitting the workload uniformly across all GPUs is better than simply picking the best fit GPU for each OLAP query. We also see that higher values of α (i.e. $\alpha \geq 0.5$) are favourable which implies that GPU acceleration is more important than PCIe overheads for this workload.

Figure 13 shows the impact of α and β on Jain's fairness index. An index value closer to 1 indicates uniform resource utilization. Lower values of β (more specifically, $\beta = 0.0$) results in fairness index above 0.9 because the scheduler only cares about scheduling the queries in such a manner so that all the devices have close to equal execution time, and thus ensuring fair resource utilization. Lower values of α (i.e. $\alpha \leq 0.5$) give slightly better fairness index values.

Using the observations from the above analysis, we choose $\alpha = 0.5$ and $\beta = 0.0$.

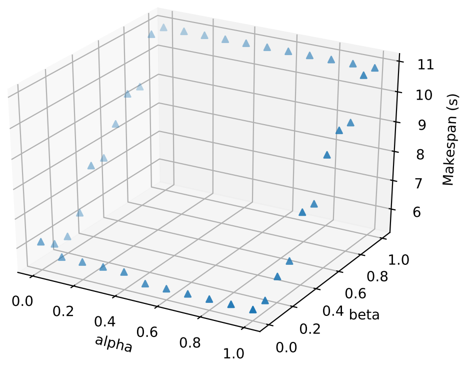


Figure 12: Impact of α *and* β *on makespan*

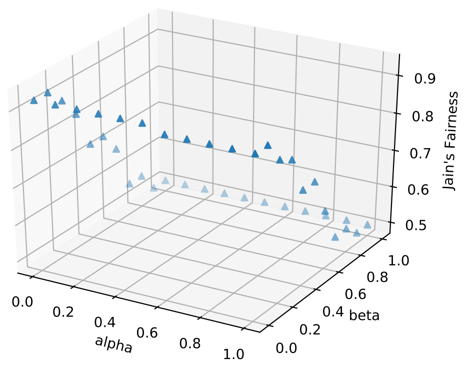


Figure 13: Impact of α *and* β *on fairness of resource utilization*

5 Future Work

There is a lot of scope for widespread adoption, and research on HTAP systems in the future. Our work tackled the problem from the scheduling aspect, and it can be further improved in several ways. The following are some of the areas that can be explored in future work:

- Use the simulator model as reference and implement a real system with EEVEE to evaluate and correlate its performance with our results.
- Alternate ways of ensuring consistency. In our work, we focused on serializability, but it would be interesting to look at methods like Snapshot Isolation.
- Extend the heterogenous cluster to include different accelerators focused on OLAP queries.
- Minimize PCIe interconnect overheads by compressing the data on the CPU end, and decompressing the data on the GPU/accelerator end.
- Our current design transfers all columns for tuples required by a query onto the GPU. It would be interesting to look at designs that transfer only specific subset of required columns and evaluate the benefits of doing this.

6 Conclusions

In this paper, we studied the benefits of using a heterogenous cluster to run HTAP queries. We proposed and implemented a novel transaction scheduler, EEVEE to efficiently route queries, and improve overall query execution latency. We also evaluated a two-level tuple replacement algorithm to manage GPU memory more effectively. Based on our evaluation, we observe that we obtain upto 4x improvement in performance compared to the baseline. Our contributions clearly show that there is tons of untapped potential, and scope for research as HTAP systems become commonplace in the near future.

References

[1] NVIDIA CUDA C/C++ Streams and Concurrency. http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf. Accessed: 2020-03-30.

[2] TPC-C Benchmark. http://www.tpc.org/tpcc/. Accessed: 2020-03-15.

[3] ABADI, D. J., BONCZ, P. A., AND HARIZOPOULOS, S. Column-oriented database systems. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1664–1665.

[4] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference* (2010), pp. 63–74.

[5] AZVINE, B., CUI, Z., NAUCK, D. D., AND MAJEED, B. Real time business intelligence for the adaptive enterprise. In *The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06)* (2006), pp. 29–29.

[6] BARTHELS, C., MÜLLER, I., SCHNEIDER, T., ALONSO, G., AND HOEFLER, T. Distributed join

algorithms on thousands of cores. *Proceedings of the VLDB Endowment 10*, 5 (2017), 517–528.

[7] Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 221–230.

[8] Bernstein, P. A., and Goodman, N. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS) 8*, 4 (1983), 465–483.

[9] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., and Zamanian, E. The end of slow networks: It's time for a redesign. *arXiv preprint arXiv:1504.01048* (2015).

[10] Blagodurov, S., Zhuravlev, S., and Fedorova, A. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS) 28*, 4 (2010), 1–45.

[11] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)* (2014), pp. 285–300.

[12] Do, J., Kee, Y.-S., Patel, J. M., Park, C., Park, K., and DeWitt, D. J. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1221–1230.

[13] Funke, F., Kemper, A., and Neumann, T. Compacting transactional data in hybrid OLTP &amp; OLAP databases. *CoRR abs/1208.0224* (2012).

[14] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 455–466.

[15] Grandl, R., Chowdhury, M., Akella, A., and Ananthanarayanan, G. Altruistic scheduling in multi-resource clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 65–80.

[16] Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 485–500.

[17] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 29–42.

[18] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), vol. 11, pp. 22–22.

[19] Hyperion Solutions Corporation. The Role of the OLAP Server in a Data Warehousing Solution.

[20] Jain, R., Chiu, D., and Hawe, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR cs.NI/9809099* (1998).

[21] Kemper, A., and Neumann, T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering* (2011), IEEE, pp. 195–206.

[22] Kepe, T. R., de Almeida, E. C., and Alves, M. A. Database processing-in-memory: an experimental study. *Proceedings of the VLDB Endowment 13*, 3 (2019), 334–347.

[23] Lepers, B., Quéma, V., and Fedorova, A. Thread and memory placement on {NUMA} systems: Asymmetry matters. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)* (2015), pp. 277–289.

[24] Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[25] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 1–16.

[26] Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 20) (Santa Clara, CA, Feb. 2020), USENIX Association.

[27] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 221–235.

[28] Oracle Corporation. Oracle Database In-Memory with Oracle Database 19c, 2019.

[29] Özcan, F., Tian, Y., and Tözün, P. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1771–1775.

[30] O'Neil, P., O'Neil, E., Chen, X., and Revilak, S. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking* (2009), Springer, pp. 237–252.

[31] Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–14.

[32] Peng, Y., Bao, Y., Chen, Y., Wu, C., Meng, C., and Lin, W. Dl2: A deep learning-driven scheduler for deep learning clusters. *arXiv preprint arXiv:1909.06040* (2019).

[33] Pohl, C., and Sattler, K.-U. Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (2018), pp. 1–10.

[34] Ramnarayan, J., Mozafari, B., Wale, S., Menon, S., Kumar, N., Bhanawat, H., Chakraborty, S., Mahajan, Y., Mishra, R., and Bachhav, K. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 2153–2156.

[35] Shanbhag, A., Madden, S., and Yu, X. A study of the fundamental performance char-acteristics of gpus and cpus for database analytics.

[36] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 491–518.

[37] van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., and Sato, M. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1541–1555.

[38] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), pp. 1–16.

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

[40] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–17.

[41] Ware, R., Mukerjee, M. K., Seshan, S., and Sherry, J. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the Internet Measurement Conference* (2019), pp. 137–143.

[42] Wu, L., Lottarini, A., Paine, T. K., Kim, M. A., and Ross, K. A. Q100: The architecture and design of a database processing unit. *ACM SIGARCH Computer Architecture News 42*, 1 (2014), 255–268.

[43] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 595–610.

[44] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (2010), pp. 265–278.

[45] Zamanian, E., Yu, X., Stonebraker, M., and Kraska, T. Rethinking database high availability with rdma networks. *Proceedings of the VLDB Endowment 12*, 11 (2019), 1637–1650.