Zhang K, Chen F, Ding X, Huai Y, Lee RB, Luo T, Wang KB, Yuan Y, and Zhang XD. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY : 1– Feb. 2015

Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources

Kai Zhang[1,2], Feng Chen[3], Member, ACM, IEEE, Xiaoning Ding[4], Yin Huai[5], Rubao Lee[2], Tian Luo[6], Kaibo Wang[2], Yuan Yuan[2], and Xiaodong Zhang[2], Fellow, ACM, IEEE

[1]*Department of Computer Science and Technology, University of Science and Technology of China, Hefei, 230027, China*
[2]*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA*
[3]*Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803, USA*
[4]*Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA*
[5]*Databricks Inc., San Francisco, CA 94105, USA*
[6]*VMware Inc., Palo Alto, CA 94304, USA*

E-mail: kay21s@mail.ustc.edu.cn; fchen@csc.lsu.edu; xiaoning.ding@njit.edu; yhuai@databricks.com; liru@cse.ohio-state.edu; tianl@vmware.com; wangka@cse.ohio-state.edu; yuanyu@cse.ohio-state.edu; zhang@cse.ohio-state.edu

Received February 21, 2015

Abstract With recent advancement on hardware technologies, new general-purpose high-performance devices have been widely adopted, such as the Graphics Processing Unit (GPU) and Solid State Drive (SSD). GPU may offer an order of higher throughput for applications with massive data parallelism, compared with the multicore CPU. Moreover, new storage device SSD is also capable of offering a much higher I/O throughput and lower latency than a traditional Hard Disk Device (HDD). These new hardware devices can significantly boost the performance of many applications; thus the database community has been actively engaging in adopting them into database systems. However, the performance benefit can not be easily reaped if the new hardwares are improperly used.

In this paper, we propose Hetero-DB, a high-performance database system by exploiting both the characteristics of the database system and the special properties of the new hardware devices in system's design and implementation. Hetero-DB develops a GPU-aware query execution engine with GPU device memory management and query scheduling mechanism to support concurrent query execution. Furthermore, with the SSD-HDD hybrid storage system, we redesign the storage engine by organizing HDD and SSD into a two-level caching hierarchy in Hetero-DB. To best utilize the hybrid hardware devices, the semantic information that is critical for storage I/O is identified and passed to the storage manager, which has a great potential to improve the efficiency and performance. Hetero-DB aims to maximize the performance benefits of GPU and SSD, and demonstrates the effectiveness for designing next generation database systems.

Keywords Databases, Heterogeneous Systems, GPU, SSD

1 Introduction

With recent advancement on new hardware technologies, new general-purpose and high-performance devices have been widely adopted, such as the computing device GPU, and new storage devices such as SSD. GPU, with unprecedentedly rich and low-cost parallel computing resources, may offer an order of magnitude higher throughput compared with a multicore CPU. In high performance computing areas, GPUs as accelerators have already been widely deployed to process performance-critical tasks. For example, according to the June 2013's Top 500 list, more than 50 supercomputers have been equipped with accelerators/co-processors (mostly NVIDIA GPUs), compared to less than 5 six years ago. As an innovative storage device, SSD has a much lower energy consumption and may achieve up to 7 times higher read throughput than a traditional hard disk. With the unprecedented computation and I/O throughput, these new general-purpose computing and storage devices can significantly improve applications' performance.

Because of GPU's high computational power and SSD's high I/O throughput, how to accelerate various workloads on GPUs and efficiently utilize SSDs have been a major research topic in many areas. Specifically, the database community has been actively engaging in making effective use of the new hardware devices to accelerate either the query execution engine or the storage engine of database systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. However, major data warehousing systems (e.g., Teradata, DB2, Oracle, SQL Server) or MapReduce-based data analytical systems (e.g., Hive, Pig, Tenzing) have not truly adopted GPUs and SSDs for productions. The reason is that the significant performance benefits from these hardwares cannot be easily reaped if they are not properly used.

To well utilize these new hardware devices, both the special properties of the hardware and the characteristics of database systems need to be taken into consideration in the database system design and implementation. For instance, in order to use GPU to maximize query execution performance, applications must fully exploit both thread-level and data-level parallelisms and well utilize SIMD (Single Instruction Multiple Data) vector units to parallelize workloads. Moreover, SSD has a fundamentally different working mechanism with HDD, where fragmentation could seriously impact performance by a factor of over 14.

Unfortunately, neither multicore-based query execution engines nor HDD-based storage engines in current database systems can well utilize the new hardware devices, resulting in resource underutilization and relatively low performance. In the following, we will show the four major technical challenges in effectively adopting GPU and SSD in query execution engines and storage engines, respectively.

1.1 GPUs as the Main Processors in Query Execution Engines

Challenge 1: A GPU-aware query execution engine. GPU has a separate device memory from the main memory, and data has to be transferred to GPU device memory via PCIe for processing. However, the PCIe transfer is expensive and is considered as the major overhead in GPU execution. The key in effectively utilizing GPU is to fundamentally understand how the two basic factors of GPU query processing are affected by query characteristics, software optimization techniques, and hardware environments. Furthermore, the programming paradigm of GPU is fundamentally different from that of CPU. Therefore, the query operators such as JOIN need to be parallelized in CUDA or OpenCL for execution on GPU, and a GPU-aware query execution engine is demanded.

Challenge 2: Efficient software infrastructure for concurrent GPU query execution. Due to the heterogeneous, data-driven characteristics of GPU operations, a single query can hardly consume all GPU resources. Dedicated query processing thus often leads to resource underutilization, which limits the overall performance of the database system. In business-critical applications such as high-performance data warehousing and multi-client dataflow analysis, a large number of users may demand query results simultaneously. As the volume of data to be processed keeps increasing, it is also essential for user queries to make continuous progress so that new results can be generated constantly to satisfy the goal of interactive analysis. The lack of concurrent querying capability restricts the adoption of GPU databases in these application fields.

Concurrent query execution consolidated usage of GPU resources enhances system efficiency and functionalities, but it makes the design of query execution engine more challenging. To achieve the highest performance, each user query tends to reserve a large amount of GPU resources. Unlike CPUs where the operating system supports fine-grained context switches and virtual memory abstractions for resource sharing, current GPU hardware and system software provide none of these interfaces for database resource management. For example, GPU tasks cannot be preempted once started; on-demand data loading is not supported during task execution; automatic data swapping service is also missing when the device memory undergoes pressure. As a result, without efficient coordination by the database, multiple GPU queries attempting to execute simultaneously can easily cause low resource usage, system thrashing, or even query abortions, which significantly degrade, rather than improve, the overall system performance. Figure 1 shows that comparing with CPU-based systems, GPU-based query execution engine lacks the support for memory management and scheduling functional facilities. Therefore, an efficient software infrastructure for supporting concurrent GPU query execution is essential for maximizing GPU utilization and throughput.

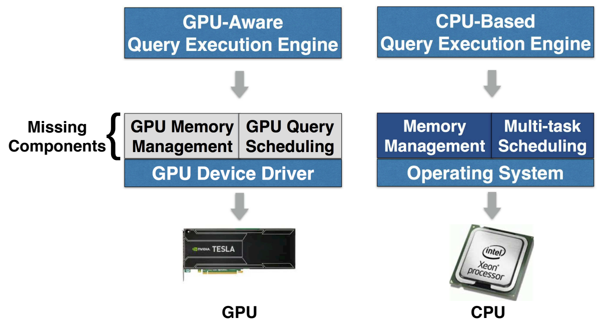


Fig. 1. Lack of software infrastructure for concurrent GPU query execution

1.2 Integrating SSD in Storage Engine

Challenge 3: The fittest position of SSDs in storage systems. Flash memory based Solid State Drive (SSD), an emerging storage technology, plays a critical role in revolutionizing the storage system design. Different from HDDs, SSDs are completely built on semiconductor chips without any moving parts. Such a fundamental difference makes SSD capable of providing one order of magnitude higher performance than rotating media, and makes it an ideal storage medium for building high-performance storage systems. However, building a storage system completely based on SSDs is often above the acceptable threshold in most commercial and daily operated systems. For example, a 32GB Intel X25-E SSD costs around $12 per GB, which is nearly 100 times more expensive than a typical commodity HDD. To build a server with only 1TB storage, 32 SSDs are needed and as much as $12,000 has to be invested in storage solely. Even considering the price-drop trend, the average cost per GB of SSDs is still unlikely to reach the level of rotating media in the near future [16]. Thus, we believe that in most systems, SSDs should not be simply viewed as a replacement for the existing HDD-based storage, but instead, SSDs should be a means to enhance it. Therefore, the fittest position of SSDs in storage systems needs to be found to strike a right balance between performance and cost.

Challenge 4: Efficient data management on hybrid storage system. Data layouts in database storage systems are established with different types of data structures, such as indexes, user tables, temporary data and others. Thus, a storage management system typically issues different types of I/O requests with different quality of service (QoS) requirements [17]. Common practice has treated storage as a black box for a long time. With a HDD-SSD hybrid storage system, the "black-box" concept of management for storage is hindering us from benefiting from the rich resources of advanced storage systems. Consequently, efficient data management on HDD-SSD hybrid storage system is required.

1.3 Our Solution

The performance of a database system depends on two major parts: the query execution engine and the storage engine. If either of them becomes the bottleneck, the overall performance of the system will be limited no matter how fast the rest part is. For example, a fast query processing engine would not improve the overall performance if the data cannot be loaded in time; and a fast storage engine would not be useful if the query execution engine is incapable of processing data with the same speed. Therefore, when designing the next generation database systems, both the query execution engine and the storage engine should be redesigned and accelerated with the most advanced hardware devices.

We propose Hetero-DB, a new database system with a complete redesign of the data processing engine and the storage engine to maximize the performance with GPU and SSD. Hetero-DB fully exploits the performance benefits from GPU and SSD, and achieves a significant speedup.

GPU Query Engine. We have designed and implemented a GPU query engine with various warehousing operators on both NVIDIA and AMD GPUs, including Selection, Join, Aggregation, and Sort. The execution engine consists of a code generator and pre-implemented query operators using CUDA/OpenCL. The code generator can generate either CUDA drive programs or OpenCL drive programs, which will be compiled and linked with pre-implemented operators.

Concurrent Query Execution. To enable multitasking on GPU processing, we propose a set of techniques including GPU device memory management and concurrent query scheduling. we present a resource management facility to support efficient executions of concurrent queries in GPU databases. It ensures high resource utilization and system performance through two key components: a query scheduler that maintains optimal concurrency level and workload on the GPUs, and a data swapping mechanism to maximize the effective utilization of GPU device memory.

Cache Hierarchy and Data Management. We organize the hybrid storage system into a two-level hierarchy. Level one, consisting of SSDs, works as a cache for level two, consisting of HDDs. To effectively utilize SSD, a direct communication channel between a DBMS and its underlying hybrid storage system is made to pass semantic information to the storage manager. We are motivated by the abundance of semantic information that is available from various DBMS components, such as the query optimizer and the execution engine, but has not been considered for database storage management. By making selected and important semantic information available to the storage manager, requests can therefore be classified into different types. With a set of predefined rules, each type is associated with a QoS policy that can be supported by the underlying storage system. At runtime, using the Differentiated Storage Services [18] protocol, the associated policy of a request is delivered to the storage system along with the request itself. Upon receiving a request, the storage system, first extracts the associated QoS policy, and then uses a proper mechanism to serve the request as required by the QoS policy.

The roadmap of this paper is as follows. Section 2 introduces the background and motivation of this research. Section 3 outlines the overall structure of Hetero-DB. Section 4 and 5 describe the GPU-aware query execution engine and the hybrid storage engine, respectively. In Section 6, we evaluate the query execution engine and the storage engine, respectively. Section 7 introduces related work, and Section 8 concludes the paper.

2 Background and Motivation

2.1 GPU as A High-Performance Computing Device

General-purpose GPUs (Graphics Processing Units), a.k.a. GPGPUs, are quickly evolving from conventional, dedicated accelerators towards mainstream commodity general-purpose computing devices. Comparing with CPU, GPU has the following three unique features: First, GPU devotes most of its die area to a large array of Arithmetic Logic Units (ALUs), and executes code in a SIMD (Single Instruction, Multiple Data) fashion. With the massive array of ALUs, GPU offers an order of magnitude higher computational throughput than CPU for applications with ample parallelism. Second, GPU has a very high memory bandwidth. NVIDIA GTX 780, for example, provides 288.4 GB/s memory bandwidth, while most recent Intel Core E5-2680v3 processor only has 68 GB/s memory bandwidth. Third, GPU effectively hides memory access latency by warp switching. Warp (or wave-front called in OpenCL), the basic scheduling unit in Nvidia GPU, can benefit zero-overhead scheduling by GPU hardware. When one warp is blocked by memory accesses, other warps whose next instruction has its operands ready are eligible to be scheduled for execution. With enough threads, memory stalls can be minimized or even eliminated.

However, utilizing GPU in accelerating database systems would face the following challenges.

1. Limited Memory Capacity and Huge Data Transfer Overhead. The capacity of GPU memory is much smaller than that of main memory. For example, the memory size of a server-class Nvidia Tesla K40 GPU is only 12 GB, while that of a data center server can be hundreds of gigabytes. Furthermore, transferring data between host memory and GPU device memory via PCIe can be a huge overhead. However, a database system generally needs to process data with a large data set. Therefore, utilizing GPUs in database systems needs a throughout evaluation to know its throughput and overheads.

2. Low Resource Utilization. A typical query execution comprises both CPU and GPU phases. The CPU phases are in charge of, e.g., initializing GPU contexts, preparing input data, setting up GPU page tables, launching kernels, materializing query results, and controlling the steps of query progress. These operations can take a notable portion of query execution time, which may cause GPU resources to be underutilized during these periods. Assuming dedicated occupation of the device, GPU queries also tend to release reserved device memory space lazily to improve data reuses and simplify algorithm implementation. This lowers the effective usage of allocated space.

3. Uncoordinated Query Co-Running. Running multiple queries on the same GPUs can improve resource utilization and system performance. Due to the lack of necessary database facilities to coordinate the sharing of GPU resources, co-running queries naively can cause serious problems such as query abortions or mediocre throughput. One of the most important functionalities not supported in current database systems is the coordination over GPU device memory usage. To maximize performance, each query tends to allocate a large amount of device memory space and keep its data on the device for efficient reuses. This causes high conflicts when multiple queries try to use device memory simultaneously. Since the underlying GPU driver does not support automatic data swapping, query co-runnings, if not managed by the database, can easily abort or suffer low performance. Even though there are recent proposals to suggest adding such service in the operating system [19], the database engine still needs to provide this functionality on its own in order to take advantage of additional information from query-level semantics for maximizing performance.

To effectively utilize GPU in database systems, we have implemented and intensively evaluated a GPU-based system prototype, and proposed a set of techniques to tackle the above issues.

2.2 Managing Heterogeneous Storage Systems

High-performance storage systems are in an unprecedented high demand for data-intensive computing in database systems. However, most storage systems, even those specifically designed for high-speed data processing, are still built on conventional HDDs with several long-existing technical limitations, such as low random access performance and high power consumption. Unfortunately, these problems essentially stem from the mechanic nature of HDDs and thus are difficult to be addressed via technology evolution. Flash memory based Solid State Drive (SSD), an emerging storage technology, plays a critical role in revolutionizing the storage system design. Different from HDDs, SSDs are completely built on semiconductor chips without any moving parts. Such a fundamental difference makes SSD capable of providing one order of magnitude higher performance than rotating media, and makes it an ideal storage medium for building high-performance storage systems.

There are two existing approaches attempting to best utilize heterogeneous storage devices. One is to rely on database administrators (DBAs) to allocate data among different devices, based on their knowledge and experiences. The other is to rely on a management system where certain access patterns are identified by runtime monitoring data accesses at different levels of the storage hierarchy, such as in buffer caches and disks.

The DBA-based approach has the following limitations: (1) It incurs a significant and increasing amount of human efforts. DBAs, as database experts with a comprehensive understanding of various workloads, are also expected to be storage experts [20]. (2) Data granularity has become too coarse to gain desired performance. As the table size becomes increasingly large, different access patterns would be imposed on different parts of a table. However, all requests associated with the same table are equally treated. (3) Data placement policies that are configured according to the common access patterns of workloads have been largely static.

Monitoring-based storage management for databases can perform well when data accesses are stable in a long term, where certain regular access patterns can be identified via data access monitoring at run time. However, monitoring-based management may not be effective under the following three conditions. First, monitoring-based methods need a period of ramp-up time to identify certain regular access patterns. For highly dynamic workloads and commonly found data with a short lifetime, such as temporary data, the ramp-up time may be too long to make a right and timely decision. Second, a recent study shows that data access monitoring methods would have difficulties to identify access patterns for concurrent streams on shared caching devices due to complex interferences [21]. Third, certain information items are access-pattern irrelevant, such as content types and data lifetime [22], which are important for data placement decisions among heterogeneous storage devices. Monitoring-based approaches would not be able to identify such information. Furthermore, monitoring-based management needs additional computing and space support, which can be expensive to obtain a deep history of data accesses.

3 Overview of Hetero-DB

Figure 2 shows the framework of Hetero-DB, where the data processing engine is redesigned with the hybrid general-purpose computing hardware devices: multi-core CPUs and GPUs. Furthermore, Hetero-DB uses SSD as a cache layer above HDD, and makes selected and important semantic information available to the storage manager. Thus, the storage engine is also redesigned. Based on the special characteristics of database systems, Hetero-DB effectively and efficiently uses the hybrid hardware devices to achieve a significant speedup.

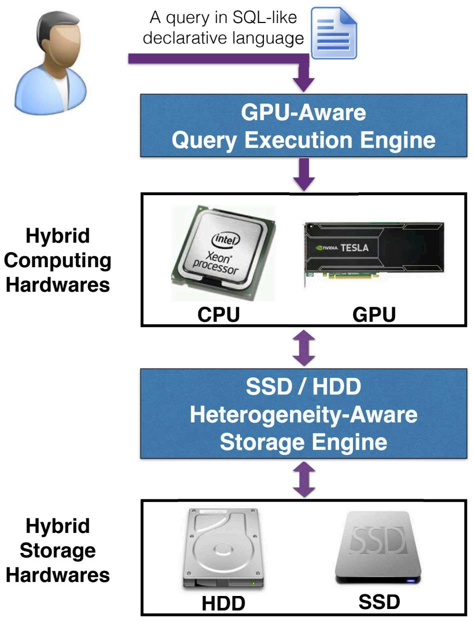

A query in SQL-like declarative language
GPU-Aware Query Execution Engine
Hybrid Computing Hardwares
CPU
GPU
SSD / HDD Heterogeneity-Aware Storage Engine
Hybrid Storage Hardwares
HDD
SSD

Fig. 2. System Overview

In the following, we will demonstrate our design of the GPU-Aware query processing engine and the heterogeneity-aware storage engine, respectively.

3.1 GPU-Aware Query Execution Engine

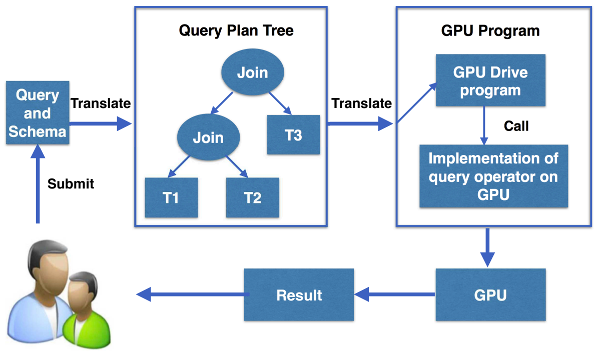


Fig. 3. Data Processing Engine

Figure 3 shows the architecture of our query engine. It is comprised of a SQL parser, a query optimizer and an execution engine. The parser and optimizer share the same codes with YSmart [23]. The execution engine consists of a code generator and pre-implemented query operators using CUDA/OpenCL. The code generator can generate either CUDA drive programs or OpenCL drive programs, which will be compiled and linked with pre-implemented operators. The engine adopts a push-based, block-oriented execution model which executes a given query plan tree in post-order sequence. It tries to keep data in GPU device memory as long as possible until all the operations on the data are finished.

To support concurrent query processing, Hetero-DB provides the functionalities needed by databases to coordinate GPU resource sharing. These functionalities enforce controls over GPU resource usage by transparently intercepting the GPU API calls from user queries. This design does not change existing programming interfaces of the underlying GPU drivers, and minimizes the modifications to the other components of the GPU query engine. The functionalities reside completely in the application space, and do not rely on any OS-level functionalities privileging to the GPU drivers.

Hetero-DB comprises two main components providing the support required for concurrent query executions. Working like an admission controller, the query scheduler component controls the concurrency level and intensity of resource contention on GPU devices. By controlling the queries that can execute concurrently at the first place, query scheduler maintains optimal workload on the GPUs that would maximize system throughput. Once a proper concurrency level is maintained, the device memory manager component further ensures system performance by resolving the resource conflicts among concurrent queries. Through VM-like automatic data swapping service, it makes sure that multiple queries with moderate resource conflicts can make concurrent progress efficiently without suffering query abortions or causing low resource utilization.

3.2 Heterogeneity-Aware Storage Engine

Figure 4 shows the architecture of the storage engine. When the buffer pool manager sends a request to the storage manager, associated semantic information is also passed. We extend the storage manager with a ”policy assignment table”, which stores the rules to assign each request a proper QoS policy, according to its semantic information. The QoS policy is embedded into the original I/O request and delivered to the storage system through a block interface. We have implemented Hetero-DB by using the Differentiated Storage Services protocol from Intel Labs [18] to deliver a request and its associated policy to a hybrid storage system. Upon receiving a request, the storage system first extracts the policy, and invokes a mechanism to serve this request.

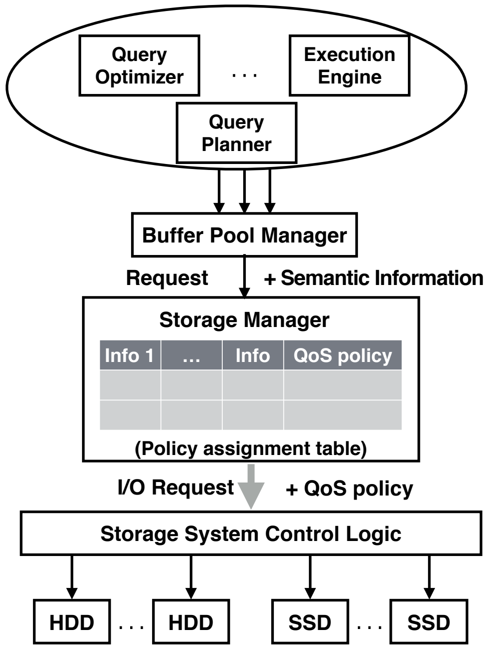


Fig. 4. Storage Engine

Compared with monitoring-based approaches [24, 25, 26], Hetero-DB has the following unique advantages: (1) Under this framework, a storage system has the accurate information of how (and what) data will be accessed. This is especially important for highly dynamic query executions and concurrent workloads. (2) A storage system directly receives the QoS policy for each request, thus could quickly invoke the appropriate mechanism to serve. (3) Besides having the ability to pass access-pattern irrelevant semantic information, in Hetero-DB, storage management does not need special data structures required by various monitoring-based operations, thus incurs no additional computation and space overhead. (4) A DBMS can directly communicate with a storage system about the QoS policy for each request in an automatic mode, so that DBAs can be relieved from the burdens of storage complexity.

Our implementation mainly involves three issues: (1) We have instrumented the query optimizer and the execution engine to retrieve semantic information embedded in query plan trees and in buffer pool requests. (2) We have augmented the data structure of the buffer pool to store collected semantic information. The storage manager has also been augmented to incorporate the ”policy assignment table”. (3) Finally, to deal with concurrency, a small region of the shared memory has been allocated for global data structures that need to be accessed by all processes.

The key to make Hetero-DB effective is associating each request with a proper QoS policy. In order to achieve our goal, we need to take the following 2 steps: 1. Understanding QoS policies and their storage implications; 2. Designing a method to determine an accurate mapping from request types to QoS policies.

In the following two sections, we will discuss the details.

4 GPU-based Data Processing Engine: Design and Implementation

4.1 Query Operators

Our query execution engine in Hetero-DB implements four operators required by star schema queries, each of which is implemented with representative algorithms based on the state-of-the-art of research.

Selection. Selection's first step is to sequentially scan all the columns in the predicates for predicate evaluation, with the result stored in a 0-1 vector. The second step is to use the vector to filter the projected columns.

Join. We implement the unpartitioned hash algorithm that has been proved to perform well for star schema queries on multi-core and many-core platforms [27, 28, 11]. We implement the hash table using both Cuckoo hash [29] and chained hash. For chained hash, hash conflicts can be avoided by making the size of hash table twice the cardinality of the input data with a perfect hash function theoretically [30]. In our study, the chained hash performs well than the Cuckoo hash. This is because star schema queries have low join selectivities, and Cuckoo hash needs more key comparisons than chained hash when there is no match for the key in the hash table.

Aggregation. We implement the hash-based aggregation which involves two steps. The first step is to sequentially scan the group-by keys and calculate the hash value for each key. The second step is to sequentially scan the hash value and the aggregate columns to generate aggregation results.

Sort. Sort operator will sort the keys first. After the keys are sorted, the results can be projected based on the sorted keys which is a gather operation. Since sort is usually conducted after aggregation, the number of tuples to be sorted is usually small which can be done efficiently through bitonic sort.

4.2 GPU Device Memory Management

The primary functionality of the device memory manager is to coordinate the conflicting demands for device memory space from different queries so that they can make concurrent progress efficiently. To achieve this goal, it relies on an optimized data swapping framework and replacement policy to minimize overhead.

4.2.1 Framework

When free device memory space becomes insufficient, instead of rejecting a query's service request, Hetero-DB tries to swap some data out from device memory and reclaim their space for better uses. This improves the utilization of device memory space and makes concurrent executions more efficient. To achieve this purpose, the device memory manager employs a data swapping framework that is motivated by a system called GDM [19]. Different from GDM, our framework resides in the application space, which cannot rely on any system-level interfaces, but has the advantage of using query-level semantics, for data swapping.

To support data swapping, the framework maintains a swapping buffer in the host memory to contain the query data that need not to reside in the device memory momentarily. When a device memory allocation request is received, it creates a virtual memory area in the swapping buffer and returns the address of the virtual memory area to the query. Device memory space only needs to be allocated when a kernel accessing the data is to be launched. The framework maintains a global list of data regions allocated on the device memory for all running queries. When free space becomes insufficient, the device memory manager selects some swappable regions from the list and evicts them to the swapping buffer. Due to the special features of multi-query workloads, several optimization techniques are employed by the framework to improve performance, as explained below.

Lazy transferring. When a query wants to copy some data to a device memory region (e.g., through cudaMemcpy in CUDA), the data are not immediately transferred to device memory until they are to be accessed in a GPU kernel. The swapping buffer serves as the temporary storage for the data to be transferred. This design prevents data from being evicted from device memory immaturely because data only need to be transferred to device memory when they are to be immediately accessed. To further reduce overhead, the memory manager marks the query source buffer copy-on-write. The data can later be transferred directly from the source buffer if it has not been changed.

Page-based coherence management. GPU queries usually reserve device memory space in large regions. The memory manager internally partitions a large region into several small, fixed-size, logical pages. Each page keeps its own state and maintains data coherence between host and device memories independently. Managing data coherence at page units has at least two performance benefits. First, by breaking a large, non-interruptible DMA operation into multiple smaller ones, data evictions can be canceled immediately when they become unnecessary (e.g., when a region is being released). Second, a partial update to a region only changes the states of affected pages, instead of a whole region, which reduces the amount of data that need to be synchronized between host and device memories.

Data reference and access advices. To avoid allocating device memory space for unused regions, the memory manager needs to know which data regions are to be referenced during a kernel execution. It is also beneficial for the memory manager to know how the referenced regions are to be accessed by a kernel. In this way, for example, the content of a region not containing kernel input data needs not to be loaded into device memory before the kernel is issued; the memory manager also need not preserve region content into the swapping buffer during its eviction if the data are not to be reused. To achieve this purpose, the memory manager provides interfaces for queries to pass data reference and access advices before each kernel invocation.

4.2.2 Data Replacement

When free device memory space becomes scarce, the memory manager has to reclaim some space for the kernel to be launched. The replacement policy that selects data regions for evictions plays an important role to system performance.

There are three main differences between data replacement in device memory and conventional CPU buffer pool management. First, the target of device memory replacement is a small number of variable-size regions rather than a large amount of uniform-size pages. GPU queries usually allocate a few device memory regions, whose sizes may differ dramatically depending on the roles of the regions and query properties. Since the physical device memory space allocated for a region cannot be partially deallocated without necessary driver support, a victim region, once selected, must be evicted from device memory completely. Second, unlike CPU databases where data evictions can be interleaved with data computation to hide the latency of replacement, a GPU kernel cannot start execution until sufficient space is vacated on the device memory for all its data sets. This makes GPU query performance especially sensitive to the latency of data replacement. Third, device memory not only has to contain input table data and output query results, but also stores various intermediate kernel objects whose content can be modified from both CPU and GPU. This makes the data access patterns of device memory regions much more diverse than buffer pool pages.

Based on these unique characteristics, we propose a policy, called CDR (Cost-Driven Replacement), that combines the effects of region size, eviction latency, and data locality to achieve good performance. When a replacement decision has to be made, CDR scans the list of swappable regions, and selects the region that would incur the lowest cost for eviction. The cost c of a region is defined in a simple formula,

$$c = e + f \times s \times l \tag{1}$$

where e represents the size of the data that needs to be evicted from device memory, s is region size, l represents the position of the region in the LRU list, and f is a constant, which we call latency factor, whose value is between 0 and 1. If two regions happen to have the same cost value, CDR breaks the tie by selecting the less recently used one for replacement.

The first part of Formula 1, e, quantifies the latency of space vacation. Its value depends on the status of the data pages in a region. For example, e is zero if none of the pages has been modified by kernels on the device memory. If some pages have been updated by the query process from the CPU, the device memory copies of those modified pages would have been invalidated and thus should not be evicted back to the swapping buffer, leading to a value of e less than s. The second part of Formula 1, $f \times s \times l$, depicts the potential overhead if the evicted region would be reused in a future kernel. The value of l is between $1/n$ and 1, depending on the region's position among the n swappable regions in the LRU order. For example, $l = 1/n$ for the least recently used region, $l = 2/n$ for the second least recently used one, and so on. The role of latency factor f is to give a heavier weight to data eviction latency in the overall cost formula.

4.3 Concurrent Query Scheduling

In an open system where user queries arrive and leave dynamically, the query scheduler maintains optimal workload on the GPUs by controlling which queries can co-run simultaneously. A query is allowed to start execution if it can make effective use of the under-utilized or unused GPU resources without incurring high overhead associated with resource contention. The GPU workload status is monitored continuously, so that delayed queries can be rescheduled as soon as enough resources become available.

A critical issue in query scheduling is to estimate the actual resource demand of a GPU query. The amount of resource being effectively utilized by a query can be much lower than its reservation. Scheduling queries based on the maximal reservation can thus cause GPUs to be under-loaded, leading to sub-optimal system throughput. In GPU databases, different queries and query phases may have diverse resource consumption, depending on query and data properties such as filter conditions, table sizes, data types, and content distributions. If the query scheduler cannot accurately predict the actual resource demand, a mistakenly scheduled query can easily bring down the overall system performance by large.

To address the problem, we propose a simple, practical metric to effectively quantify the resource demand of a GPU query. The design of the metric is based on some observations that are generally applicable to analytical GPU databases. First, for GPU query processing, the utilization of device memory space has the principal impact on system throughput and can be frequently saturated under multi-query workloads. Unlike compute cycles and DMA bandwidth that can be freely reused, reusing a device memory region requires data evictions and space re-allocation, which can potentially incur high overhead. This makes system performance strongly correlated with the demand for and utilization of device memory space. Second, to ensure data transfer and kernel execution efficiencies, analytical GPU engines usually employ a batch-oriented, operator-based query execution scheme. Under this scheme, table data are partitioned into large chunks and pushed from one operator to another for processing. It is thus a good model to consider query execution as a temporal sequence of operators, each of which accepts an input data chunk, processes it with the GPU, and generates an output data chunk that may be passed to the next operator for further processing.

Based on the above observations, we define a metric called weighted device memory demand, or briefly weighted demand, which is the weighted average of the device memory space consumed by a query's operator sequence. The weight is computed as the percentage of query execution time spent in each operator. The device memory space consumed by an operator equals the maximal total size of device memory regions referenced by any GPU kernel in the operator. Suppose that each operator's execution time and device memory consumption are ti and mi respectively, the weighted demand m of the query can be computed by

$$m = \frac{\Sigma(t_i \times m_i)}{\Sigma t_i} \tag{2}$$

The device memory consumption of an operator can be computed from query predicates and table data statistics. The execution time of an operator can be predicted through modeling, as has been shown in our previous work [31].

To accommodate the changes of resource consumption in different query phases, the weighted demand of a query is dynamically updated as the query executes, and is exposed to the query scheduler for making timely scheduling decisions. When a new query arrives, the query scheduler computes its initial weighted demand. If the number exceeds the available device memory capacity, which is measured by the difference between the device memory capacity and the sum of weighted demands of scheduled queries, the query's execution needs to be delayed. The query scheduler considers rescheduling a postponed query every time when a running query's resource demand changes or when a query finishes execution.

5 Utilizing SSD for High Performance I/O

In order to turn our design of the heterogeneous-aware storage engine into a reality, we must address the following technical issues.

Associating a proper QoS policy to each request: Semantic information does not directly link to proper QoS policies that can be understood by a storage system. Therefore, in order for a storage system to be able to serve a request with the correct mechanism, we need a method to accomplish the effective mapping from semantic information to QoS policies. However, a comprehensive solution must systematically consider multiple factors, including the diversity of query types, the complexity of various query plans, and the issues brought by concurrent query executions.

Implementation: Two challenges need to be addressed. (1) The QoS policy of each request eventually needs to be passed into the storage system. A DBMS usually communicates with storage through a block interface. However, current block interfaces do not allow passing anything other than the physical information of a request. (2) A hybrid storage system needs an effective mechanism to manage heterogeneous devices, so that data placement would match the unique functionalities and abilities of each device. Data also needs to be dynamically moved among devices to respond access pattern changes.

We will first discuss general QoS policies and then introduce the specific policies used in this paper.

5.1 Overview of QoS Policies

QoS policies provide a high-level service abstraction for a storage system. Through a set of well defined QoS policies, a storage system can effectively quantify its capabilities without exposing device-level details to users.

A QoS policy can either be performance related, such as latency or bandwidth requirements, or non-performance related, such as reliability requirements. All policies of a storage system are dependent on its hardware resources and organization of these resources. For example, if a storage system provides a reliability policy, then for an application running on such a storage system, when it issues write requests of important data, it can apply a reliability policy to these requests. Thus, when such a request is delivered, the storage system can automatically replicate received data to multiple devices.

On the one hand, QoS policies can isolate device-level complexity from applications, thus reducing the knowledge requirement on DBAs, and enabling heterogeneity-aware storage management within a DBMS. On the other hand, these policies determine the way in which a DBMS can manage requests. It is meaningless to apply a policy that cannot be understood by the storage system. Therefore, a different storage management module may be needed if a DBMS is ported to another storage system that provides a fundamentally different set of policies.

5.2 QoS Policies of a Hybrid Storage System

We will demonstrate how to enable automatic storage management with a case study where the QoS policies are specified as a set of *caching priorities.*

The underlying system is a hybrid storage system prototype organized into a two-level hierarchy. The first level works as a cache for the second level. We use SSDs at the first level, and HDDs at the second level. In order to facilitate the decision making on cache management (which block should stay in cache, and what should not), its QoS policies are specified as a set of caching priorities, which can be defined as a 3-tuple:

$\{N, t, b\}$, where $N > 0$, $0 \leq t \leq N$, and $0\% \leq b \leq 100\%$.

Parameter N defines the total number of priorities, where a smaller number means a higher priority, i.e., a better chance to be cached.

Parameter t is a threshold for "non-caching" priorities: blocks accessed by a request of a priority $\geq t$ would have no possibility of being cached. In this paper, we set $t = N - 1$. So, there are two non-caching priorities, $N - 1$ and N. We call priority $N - 1$ "*non-caching and non-eviction*", and call N "*non-caching and eviction*".

There is a special priority, called write buffer, configured by parameter b. More details about these parameters will be discussed later.

For each incoming request, the storage system first extracts its associated QoS policy, and then adjusts the placement of all accessed blocks accordingly. For example, if a block is accessed by a request associated with a "high priority", it will be fetched into cache if it is not already cached, depending on the relative priority of other blocks that are already in cache. Therefore in practice, the priority of a request is eventually transformed to the priority of all accessed data blocks. In the rest of paper, we will also use "priority of a block" without further explanation.

5.3 QoS Policy For Each Request

5.3.1 Request Types

A database I/O request has various semantic information. For the purpose of caching priorities, in this paper, we consider semantic information from the following categories.

Content type: We focus on three major content types: regular table, index and temporary data. Regular tables define the content of a database. They are the major consumers of database storage capacity. Indexes are used to speedup the accessing of regular tables. Temporary data, such as a hash table [8], would be generated during the execution of a query, and removed before the query is finished.

Access pattern: It refers to the behavior of an I/O request. It is determined by the query optimizer. A table may be either sequentially scanned or randomly accessed. An index is normally randomly accessed.

According to collected semantic information, we can classify requests into the following types: (1) sequential requests; (2) random requests; (3) temporary data requests; (4) update requests. The discussion of QoS policy mapping will be based on these types.

5.3.2 Policy Assignment in a Single Query

We will present five rules that are used to map each request type to a proper QoS policy which, in this case study, is a caching priority. For each request, the rules mainly consider two factors: 1) performance benefit if data is served from cache, and 2) data reuse possibility. These two factors determine if we should allocate cache space for a disk block, and if we decide to allocate, how long should we keep it in cache. In this subsection, we will consider priority assignment within the execution of a single query, and then discuss the issues brought by concurrent query executions in the next subsection.

1. Sequential Requests. In our storage system, the caching device is an SSD, and the lower level uses HDDs, which can provide a comparable sequential access performance to that of SSDs. Thus, it is not beneficial to place sequentially accessed blocks in cache.

RULE 1: *All sequential requests will be assigned the "non-caching and non-eviction" priority.*

A request with the "non-caching and non-eviction" priority has two implications: (1) If the accessed data is not in cache, it will not be allocated in cache; (2) If the accessed data is already in cache, its priority, which is determined by a previous request, will not be affected by this request. In other words, requests with this priority do not affect the existing storage data layout.

2. Random Requests. Random requests may benefit from cache, but the eventual benefit is dependent on data reuse possibility. If a block is randomly accessed once but never randomly accessed again, we should not allocate cache space for it either. Our method to assign priorities for random requests is outlined in Rule 2.

RULE 2: *Random requests issued by operators at a lower level of its query plan tree will be given a higher caching priority than those that are issued by operators at a higher-level of the query plan tree.*

This rule can be further explained with the following auxiliary descriptions.

Level in a query plan tree: For a multi-level query plan tree, we assume that the root is on the highest level; the leaf that has the longest distance from the root is on the lowest level, namely Level 0.

Related operators: This rule relates to random requests that are mostly issued by "index scan" operators. For such an operator, the requests to access a table and its corresponding index are all random.

Blocking operators: With a blocking operator, such as hash or sorting, operators at higher levels or its sibling operator cannot proceed unless it finishes. Therefore, the levels of affected operators will be recalculated as if this blocking operator is at Level 0.

Priority range: Note that there are totally N different priorities, but not all of them will be used for random requests. Instead, random requests are mapped to a consecutive priority range $[n_1, n_2]$, where $n_1 \leq n_2$. So, n_1 is the highest available priority for random requests, and n_2 is the lowest available priority.

When multiple operators access the same table: For some query plans, the same table may be randomly accessed by multiple operators. In this case, the priorities of all random requests to this table are determined by the operator at the lowest level of the query plan tree. If there is an operator that sequentially accesses the same table, the priority of this operator's requests is still determined by Rule 1.

Function (3) formalizes the process of calculating the priority of a random request, issued by an operator at Level i of the query plan tree. Assume l_{low} is the lowest level of all random access operators in the query plan tree, while l_{high} is the highest level. And L_{gap} represents this gap, where $L_{gap} = l_{high} - l_{low}$. Assume C_{prio} is the size of the available priority range $[n1, n2]$, so $C_{prio} = n_2 - n_1$.

$$p_i = \begin{cases} n_1 & if C_{prio} = 0 \\ n_1 & if L_{gap} = 0 \\ n_1 + i - l_{low} & if C_{prio} \geq L_{gap} \\ n_1 + \lfloor C_{prio} \times \frac{(i - l_{low})}{L_{gap}} \rfloor & if C_{prio} < L_{gap} \end{cases} \tag{3}$$

The last branch of this function describes the case when a tree has too many levels that there are not enough priorities to assign for each level. In this case, we can assign priorities according to the relative location of operators, and operators at neighboring levels may share the same priority.

Let us take the query plan tree in Figure 5 as an example. In this example, three tables are accessed: *t.a*, *t.b* and *t.c*. We assume that the available priority range is [2,5]. Both operators that access *t.a* are index scans. Since the lowest level of random access operators for *t.a* is Level 0, all random requests to *t.a* and its index would be assigned Priority 2. It also means that requests from the "index scan" operator at Level 1 are assigned the same priority: Priority 2.

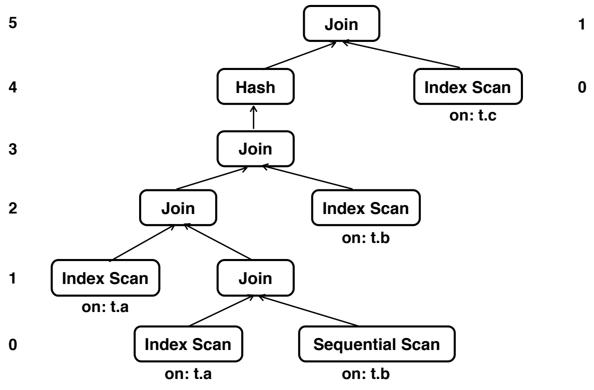


Fig. 5. Example query plan tree. This tree has 6 levels. Root is on the highest level: Level 5. Due to the blocking operator "hash", the other two operators on Level 4 and 5 are re-calculated as on Level 0 and 1.

As to *t.b*, there are also two related operators. But according to Rule 1, all requests from the "sequential scan" operator (Level 0) are assigned the "non-caching and non-eviction" priority. Requests from the other operator (Level 2) that accesses t.b are random. According to Function (3), requests from this operator should be assigned Priority 4.

For Table *t.c*, it is accessed by random requests from an "index scan" operator. However, due to the blocking operator "hash" on Level 4, the "index scan" operator is considered at Level 0 in priority recalculation, and thus all random requests to table *t.c* would be assigned Priority 2.

3. Temporary Data Requests. Queries with certain operators may generate temporary data during execution. There are two phases associated with temporary data: *generation phase* and *consumption phase.* During generation phase, temporary data is created by a write stream. During consumption phase, temporary data is accessed by one or multiple read streams. In the end of consumption phase, the temporary data is deleted to free up disk space. Based on this observation, we should cache temporary data blocks once they are generated, and immediately evict them out of cache at the end of their lifetime.

RULE 3: *All read/write requests to temporary data are given the highest priority. The command to delete temporary data is assigned the "non-caching and eviction" priority.*

A request with the "non-caching and eviction" priority has two implications: (1) If the accessed data is not in cache, it will not be promoted into cache; (2) If the accessed data is already in cache, its priority will be changed to "non-caching and eviction", and can be evicted timely. Thus, requests with the "non-caching and eviction" priority only allow data to leave cache, instead of getting into cache.

Normally, if a DBMS is running with a file system, the file deletion command only results in metadata changes of the file system, without notifying the storage system about which specific blocks have become useless. This becomes a problem because temporary data may not be evicted promptly. And because of its priority, temporary data cannot be replaced by other data. Gradually, the cache will be filled with obsolete temporary data.

This issue can be addressed by the newly proposed TRIM command [1], which can inform the storage system of what LBA (logical block address) ranges have become useless due to file deletions, or other reasons. Supported file systems, such as EXT4, can automatically send TRIM commands once a file is deleted. For a legacy file system that does not support TRIM, we can use the following workaround to achieve the same effect: Before a temporary data file is deleted, we issue a series of read requests, with the "non-caching and eviction" priority, to scan the file from beginning to end. This will in effect tell the storage system that these blocks can be evicted immediately. This workaround incurs some overhead at an acceptable level, because the read requests are all sequential.

Update Requests We allocate a small portion of the cache to buffer writes from applications, so that they do not access HDDs directly. With a write buffer, all written data will first be stored in the SSD cache, and flushed into the HDD asynchronously. Therefore, we apply the following rule for update requests:

RULE 4: *All update requests will be assigned the "write buffer" priority.*

There is a parameter b that determines how much cache space is allocated as a write buffer. When the occupied space of data written by update requests exceeds b, all content in the write buffer is flushed into HDD. Note that the write buffer is not a dedicated space. Rather, it is a special priority that an update request can "win" cache space over requests of any other priority. For OLAP workloads in this paper, we set b at 10%.

5.3.3 Concurrent Queries

When multiple queries are co-running, I/O requests accessing the same object might be assigned different priorities depending on which query they are from. To avoid such non-deterministic priority assignment, we apply the following rule for concurrent executions.

RULE 5:

1. *For sequential requests, temporary data requests and updates, the priority assignment still follows Rule 1, Rule 3 and Rule 4;* 2. *for random requests that access the same object (table or index) but for different queries, they are assigned the highest of all priorities, each of which is determined by Rule 3 and independently based on the query plan of each running query;*

To implement this rule, we store some global information for all queries to access: a hash table $H < oid, list >$, two variables gl_{low} and gl_{high}.

- The key "*oid*" stores an object ID either of a table or of an index.
- The structure "*list*" for each *oid* is a list.
- Each element of *list* is a 2-tuple $< level, count >$. It means that among all queries, there are totally *count* operators accessing *oid*, and all of these operators are on Level *level* in their own query plan tree. If some operators on different levels of a query plan tree are also accessing *oid*, we need another element to store this information.
- Variable gl_{low} (the global lowest level of all random operators) stores the minimum value of all l_{low} according to each query; similarly, gl_{high} stores the maximum value of all l_{high} according to each query.

All these data structures are updated upon the start and end of each query. To calculate the priority of a random request, with concurrency in consideration, we can still use Function (3), just changing l_{low} into gl_{low}, and similarly l_{high} to gl_{high}, and thus L_{gap} would be $gl_{high} - gl_{low}$. Figure 6 describes this process.

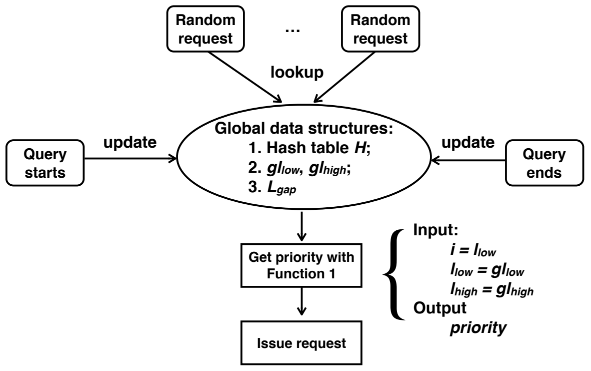


Fig. 6. The process to calculate request priorities

Table 1 summarizes all the rules hStorage-DB uses to assign caching priorities to requests.

Table 1. Rules to assign priorities

Request type	Priority	Rule
Temporary data requests	1	Rule 3
random requests	2 ... N-2	Rules 2, 5
sequential requests	N-1	Rule 1
TRIM to temporary data	N	Rule 3
updates	write buffer	Rule 4

6 Experiments

In this section, we evaluate the performance of the GPU-aware query execution engine and the hybrid HDD/SSD based storage engine. To show the effectiveness of our query execution engine and the storage engine, they are evaluated separately. In the evaluation of the GPU-Aware query execution engine, instead of reading data from SSD or HDD, we transfer data from the main memory to the GPU device memory. When we are evaluating the HDD/SSD heterogeneity-aware storage engine, the CPU-based query execution engine is used for a fair comparison.

6.1 Performance of GPU-Aware Query Execution Engine

We conduct the experiments on NVIDIA GTX 680. The experiments are conducted with NVIDIA Linux driver 310.44 with CUDA SDK 5.0.35. We use the Star Schema Benchmark (SSBM) which has already been widely used in various data warehousing research studies. It has one fact table lineorder and four dimension tables date, supplier, customer, part, which are organized in a star schema fashion. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments, we run the benchmark with a scale factor of 10 which will generate the fact table with 60 million tuples.

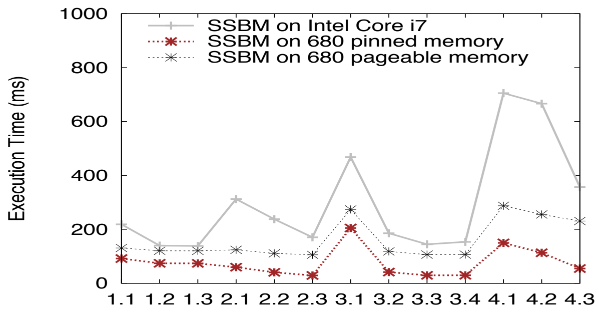


Fig. 7. SSBM performance comparison. For the performance on Intel Core i7, the performance of Q4.1 and Q4.2 are the performance on OpenCL engine while the rest are the performance on MonetDB.

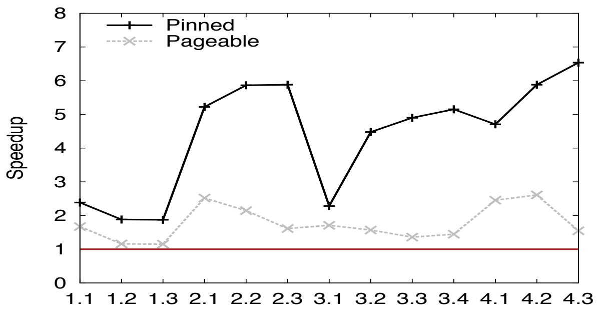


Fig. 8. SSBM performance speedup over CPU

Figure 7 shows the execution time of SSBM queries on our GPU-aware query execution engine, and Figure 8 shows the performance speedup of GPU over CPU. Our comparisons of CPU and GPU query execution engine are based on the following two kinds of performance numbers. First, the GPU performance is the performance of the CUDA engine on NVIDIA GTX 680. Second, the CPU performance for each query is the better one between the performance of MonetDB and of our OpenCL query engine on Intel Core i7. We conduct the experiments under two conditions: 1) data are available in the pinned memory; and 2) data are available in the pageable memory.

For Q1.1 to Q1.3 and Q3.1, processing on GPU can only gain around 2x speedup, as is shown in Figure 8. Queries in flight 1 are dominated by selection operations. They cannot benefit from the transfer overlapping technique. Although data compression technique can reduce the PCIe transfer overhead, the kernel execution performance cannot be improved. Since selection does not involve much computation, processing on GPU will not have significant performance speedup. Q3.1 is dominated by the random accesses of data from dimension tables. It cannot benefit much from both the data compression technique and the transfer overlapping technique. Furthermore, the random accesses cannot effectively utilize the bandwidth of GPU device memory. In this case, we cannot gain significant performance speedup.

We have the following three findings:

- The GPU query engine outperforms the CPU query engine for processing all SSBM queries. However, the performance speedup varies significantly depending on query characteristics and system setups.
- The key to obtain high query execution performance on GPU is to prepare the data in the pinned memory, where 4.5x-6.5x speedups can be observed for certain queries. When data are in the pageable memory, the speedups are only 1.2x-2.6x for all SSBM queries.
- GPU has limited speedups (around 2x) for queries: 1) dominated by selection operations, and 2) dominated by random accesses to dimension tables caused by high join selectivities and projected columns from dimension tables.

6.2 Performance of Concurrent Query Execution

We call our system functionality for concurrent query execution as MultiQx-GPU. For comparison, we call the baseline system as YDB. Through coordinated sharing of GPU resources, MultiQx-GPU improves system throughput by letting multiple queries make efficient progress concurrently. In this subsection we evaluate the overall performance of MultiQx-GPU in supporting concurrent executions. The evaluation is performed by co-running SSB queries pair wisely. Among 91 possible query combinations, we select 69 pairs of co-runnings whose peak device memory consumption exceeds device memory capacity (i.e., suffering conflicts). We measure their throughput achieved with MultiQx-GPU, and compare them with the original YDB system. The first two bars of each group in Figure 9 show the results.

It can be seen that, by processing multiple queries at the same time, MultiQx-GPU greatly enhances system performance compared with dedicated query executions. The throughput is consistently improved across all 69 co-runnings, by an average of 39% (at least 15%) as compared with YDB. For the co-runnings of q32 with q41, q33 with q41, and q43 with itself, the improvements are more than 55%. The high performance leap achieved by MultiQx-GPU is mainly attributed to the better utilization of GPU resources. Under the efficient management of MultiQx-GPU, the DMA bandwidth and GPU computing cycles unused by one query can be allocated to serve the resource requirements from other queries. The efficient utilization of resources improves overall system throughput.

6.3 Experiments with Replacement Policies

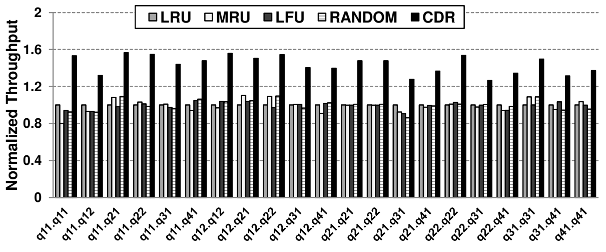


Fig. 10. Performance of co-running selected SSB queries under various data replacement policies. Throughput is normalized against LRU.

By controlling the selection of proper victim regions to evict under resource contention, data replacement policy plays an important role to system throughput. This subsection presents the results of our experiment with several data replacement policies to support multi-query executions and verifies the effectiveness of CDR in improving MultiQxGPU performance. We compare the performance of five replacement policies, LRU (Least Recently Used), MRU (Most Recently Used), LFU (Least Frequently Used), RANDOM, and CDR. The first four policies are selected because they are widely used in conventional multitasking and data management systems. We measure the throughput of the same workloads used in the previous two subsections, achieved using MultiQx-GPU (all optimizations are enabled) with different replacement policies. Due to space constraint, we randomly select 6 queries and only present the results for their co-runnings, but similar observations can be made with other queries as well.

As shown in Figure 10, there are no significant differences among the performance of LRU, MRU, LFU, and RANDOM; they perform unevenly, but closely match each other under different workloads. CDR, however, performs much better than other policies across all query co-runnings, consistently improving system throughput by 44% on average (56% at max-

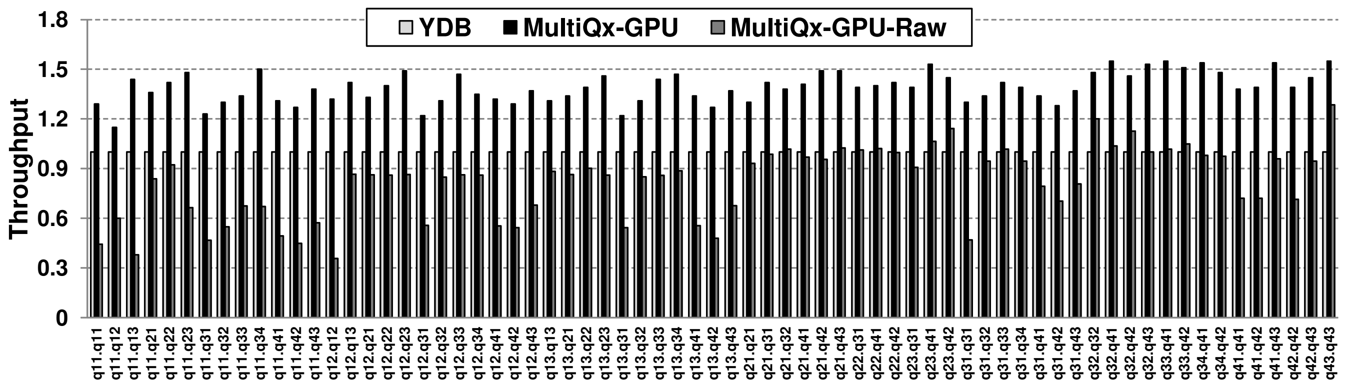


Fig. 9. Throughput of pairwise SSB query co-runnings in three different systems. MultiQx-GPU-Raw is a variant of MultiQx-GPU without optimizations. p.q denotes the combination of queries p and q.

imum) compared with LRU. The performance advantage of CDR compared with other policies is expected, due to its careful design to select victim regions that minimize space eviction and data swapping costs. On the contrary, the other four policies do not consider the unique features of GPU queries and their concurrent executions. The criteria they use to make replacement decisions are rather random in terms of the benefits to overall system performance, often leading to increased kernel launch latency and unnecessary data swapping.

6.4 Performance of Heterogeneity-aware Storage Engine

Our experiment platform consists of two machines, connected with a 10 Gb Ethernet link. One machine runs the DBMS, and the other is a storage server. Both are of the same configurations: two Intel Xeon E5354 processors, each having four 2.33GHz cores, 8 GB of main memory, Linux 2.6.34, two Seagate Cheetah 15.7K RPM 300 GB HDDs. The storage server has an additional Intel 320 Series 300 GB SSD to be our cache device. We choose TPC-H at a scale factor of 30 as our OLAP benchmark. With 9 indexes, the total dataset size is 46GB.

Classification is meaningful only if a DBMS issues I/O requests of different types. In order to verify this assumption, for each query, we run it once, and count the number of I/O requests of each type, as well as the total number of disk blocks served for requests of each type. As shown in Figure 11, we can observe requests of various types in TPC-H: sequential requests (Seq), random requests (Rand), and temporary data requests (Tmp).

For each query, we run it with the following four different storage configurations. (1) HDD-only; (2) LRU; (3) hStorage-DB; (4) SSD-only. HDD-only shows the baseline case when all I/O requests are served by a hard disk drive; SSD-only shows the ideal case when all I/O requests are served by an SSD; LRU emulates a classical approach when cache is managed by the LRU (least recently used) algorithm; Since we only evaluate the heterogeneity-aware storage engine, the queries are all run on CPU instead of GPU. hStorage-DB is used to represent such the Hetero-DB with only the heterogeneity-aware storage engine. When we experiment with LRU and hStorage-DB, the SSD cache size is set to be 32GB, unless otherwise specified.

6.4.1 Sequential Requests

To demonstrate the ability of hStorage-DB to avoid unnecessary overhead for allocating cache space for low-locality data, we have experimented with Queries 1, 5, 11 and 19, whose executions are dominated by sequential requests, according to Figure 11. Test results are shown in Figure 12.

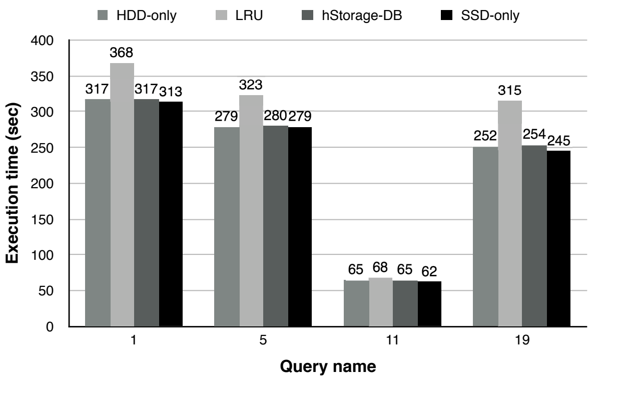
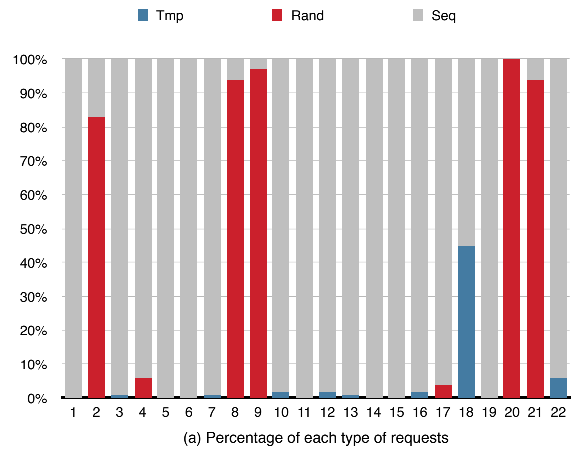


Fig. 12. Execution time of queries dominated by sequential requests.

There are three observations from Figure 12 (1) The advantage of using SSD is not obvious for these



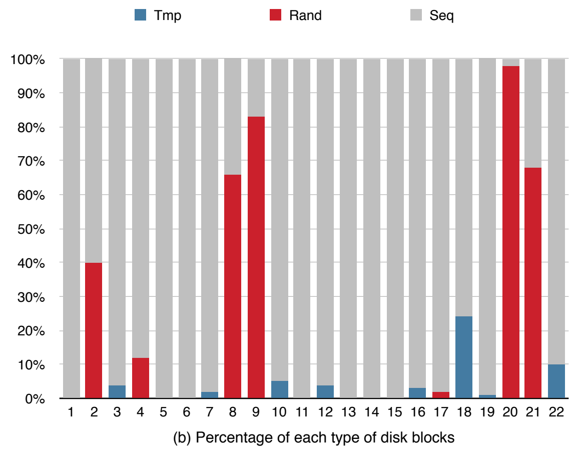


Fig. 11. Diversity of IO requests in TPC-H queries. X-axis: Name of queries in TPC-H. Y-axis: Percentage of each type.

queries. (2) If the cache is managed by LRU, which is not sensitive to sequential requests, the overhead can be significant. For example, compared with the baseline case, the execution time of LRU cache increased from 317 to 368 seconds for Q1, and from 252 to 315 seconds for Q19, resulting in a slowdown of 16% and 25% respectively. (3) Within the framework, sequential requests are associated with the "non-caching and non-eviction" priority, so they are not allocated in cache, and thus incur almost no overhead.

6.4.2 Random Requests

In order to show the effectiveness of Rule 2. we have experimented with Q9 and Q21. Both have a significant amount of random requests, according to Figure 11. Results are plotted in Figure 13.

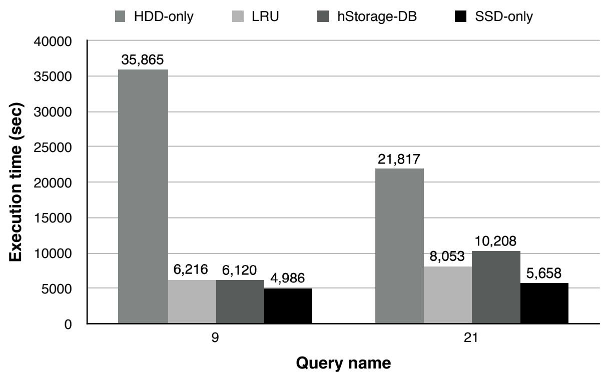


Fig. 13. Execution time of queries dominated by random requests.

We have three observations from Figure 13. (1) The advantage of using SSD is obvious for the such queries. The performance of the ideal case (SSD-only) is far more superior than the baseline case (HDD-only). For Q9 and Q21, the speedup is 7.2 and 3.9 respectively. (2) Both queries have strong locality. LRU can effectively keep frequently accessed blocks in cache, through its monitoring of each block, and hStorage-DB achieves the same effect, through a different approach. (3) For Q21, hStorage-DB is slightly lower than LRU, which will be explained later.

6.4.3 Temporary Data Requests

In this section, we demonstrate the effectiveness of Rule 3 by experimenting with Q18, which generates a large number of temporary requests during its execution. Results are plotted in Figure 14.

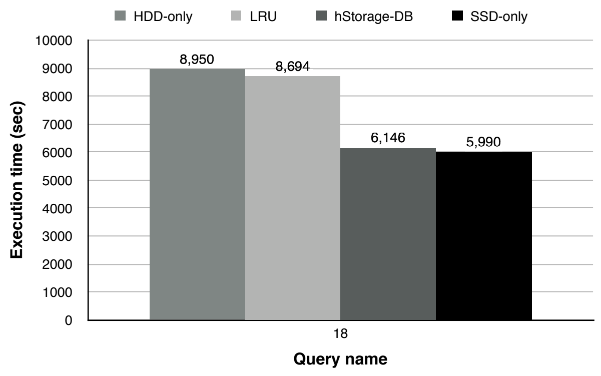
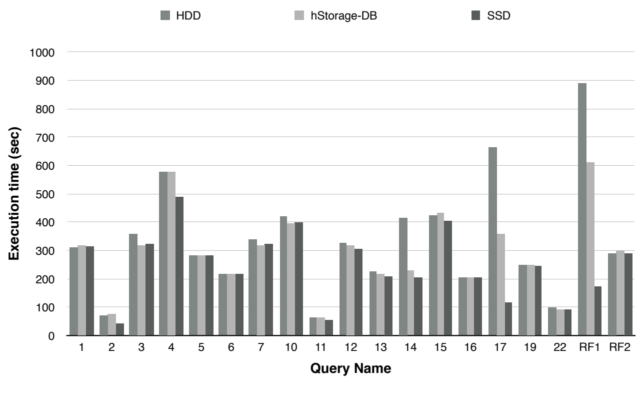
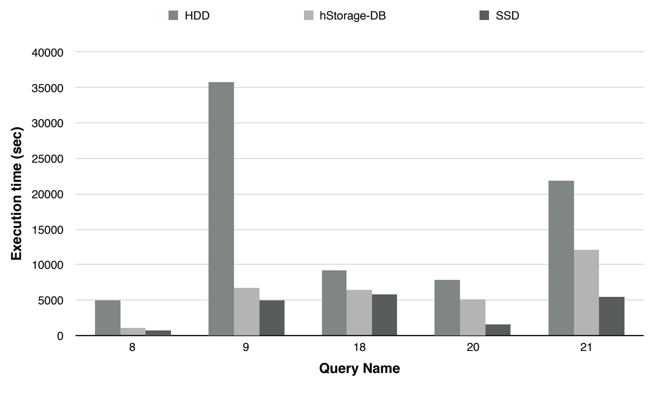


Fig. 14. Execution time of Query 18.

This figure gives us the following three observations. (1) The advantage of using SSD is also obvious for this query, giving a speedup of 1.45 over the baseline case (HDD-only). (2) LRU also improves performance, because some temporary data blocks can be cached. But they are not kept long enough, so the



(a) Short queries



(a) Long queries

Fig. 15. Execution time of queries when packed into one query stream.

speedup is not obvious. (3) hStorage-DB achieves a significant speedup because temporary data is cached until the end of its lifetime.

6.4.4 Running a Sequence of Queries

We have tested the performance of hStorage-DB with a stream of "randomly" ordered queries. We use the order of power test by the TPC-H specification. Results are shown in Figure 15. In this figure, "RF1" is the update query at the beginning, and "RF2" is the update query in the end. For readability, the results of short queries and those of long queries are shown separately. According to the results, hStorage-DB shows clear performance improvements for most queries.

This experiment with a long sequence of queries took hStorage-DB more than 10 hours to finish, which shows the stability and practical workability of the hStorage-DB solution. Different from running each query independently, the success of this experiment involves the issues of data reuse and cache space utilization during a query stream. Particularly, useless cached data left from a previous query need to be effectively evicted from cache. Experimental results have shown that in the framework of hStorage-DB, (1) temporary data can be evicted promptly, by requests with the "non-caching and eviction" priority; and (2) data randomly accessed by a previous query can be effectively evicted by random requests of the next query, if it will not be used again.

7 Related Works

7.1 Heterogeneous Query Execution Engine

The use of GPUs for database applications has been intensively studied in existing research. Some works focus on the designs and implementations of efficient GPU algorithms for common database operations such as join [3, 11], selection [2], sorting [2, 6], and spatial computation [1, 15], achieving orders of magnitude of performance improvement over conventional CPU-based solutions. Other works exploit various software optimization techniques to accelerate query plan generations [32], improve kernel execution efficiency [8, 13], reduce PCIe data transferring [13, 33], and support query co-processing with both GPUs and CPUs [34].

Our work in this paper is mainly related to recent development efforts of GPU query engines. Ocelot [35] is a hybrid OLAP query processor as an extension for MonetDB. By adopting a hardware-independent query engine design, it supports efficient executions of OLAP queries on both CPUs and GPUs. Ocelot provides a memory management interface that abstracts away the details of the underlying memory structure to support portability. The memory manager can also perform simple data swapping within a single query. However, it does not have sufficient mechanisms or policies to support correct, efficient executions of queries in concurrent settings. Different from Ocelot, our work supports concurrent query execution on GPU with a query scheduler and a memory manager with Cost-Driven Replacement policy. MapD is a spatial database system using GPUs as the core query processing devices. Through techniques such as optimized spatial algorithm implementations, kernel fusing, and data buffering, MapD outperforms existing CPU spatial data pro-

http://www.mapd.com/

cessing systems by large margins. GPUTx [9] is a high-performance transactional GPU database engine. It batches multiple transactional queries into the same kernel for efficient executions on the GPU and ensures isolation and consistency under concurrent updates. The workloads GPUTx targets are short-running, small tasks that would not cause device memory contention. The techniques thus cannot be used for concurrent analytical query processing on GPUs, where tasks usually have long time spans and have high demands for device memory space. HyPE [36] is a hybrid engine for CPU-GPU query co-processing. The idea of its operator-based execution cost model is similar to the weighted demand metric proposed in this paper. Compared with these studies, Hetero-DB identifies the critical demands and opportunities of supporting concurrent query executions in analytical GPU databases. It addresses a set of issues in GPU resource management to achieve high system performance under multi-query workloads.

In addition, there are several research works on GPU resource management in general-purpose computing systems. PTask [37] adds abstractions of GPUs in the OS kernel to support managing GPUs as first-class computing resources. It provides a dataflow-based programming model and enforces system-level management of GPU computing resources and data movement. TimeGraph [38] is GPU scheduler to provide performance isolation for real-time graphics applications. Both PTask and TimeGraph target computation-bond workloads where memory usage will not exceed the capacity of GPU memory. Therefore, they do not have any facilities for memory management and do not take memory swapping cost into consideration in the scheduling policy. However, database queries generally involve processing a huge amount of data which may incur a high cost in PCIe data transfer and should be the major factor in making scheduling policy. Moreover, as the data size for multiple queries may exceed the capacity of GPU memory, GPU memory management facility is also highly demanded. Gdev [39] is an open-source CUDA driver and runtime system. It supports inter-process communication through GPU device memory and provides simple data swapping functionality based on the IPC mechanism. GDM [19] is an OS-level device memory manager, which motivated the design of Hetero-DB's data swapping framework.

7.2 Heterogeneous Storage Engine

There have been several different approaches to managing storage data in databases, each of which has unique merits and limits. The most classical approach is to apply a replacement mechanism to keep or evict data at different levels of the storage hierarchy. Representative replacement algorithms include LIRS [26] that is used in MySQL and other data processing systems, and ARC [40] that is used in IBM storage systems and ZFS file system. The advantage of this approach is that it is independent of workloads and underlying systems. The nature of general-purpose enables this approach to be easily deployed in a large scope of data management systems. However, the two major disadvantages are (1) this approach would not take advantage of domain knowledge even it is available; and (2) it requires a period of monitoring time before determining access patterns for making replacement decisions.

There are two recent and representative papers focusing on SSD management in database systems. In [24], authors propose an SSD buffer pool admission management by monitoring data block accesses to disks and distinguishing blocks into warm regions and cold regions. A temperature-based admission decision to the SSD is made based on the monitoring results: admitting the warm region data and randomly accessed data to the SSD, and making the cold region data stay in hard disks. In [7], authors propose three admission mechanisms to SSDs after the data is evicted from memory buffer pool. The three alternative designs include (1) clean write (CW) that never writes the dirty pages to the SSD; (2) dual-write (DW) that writes dirty pages to both the SSD and hard disks; and (3) lazy-cleaning (LC) that writes dirty pages to the SSD first, and lazily copies the dirty pages to hard disks. Although specific to the database domain, this approach has several limitations that are addressed by hStorage-DB.

Compared the aforesaid prior work, hStorage-DB leverages database semantic information to make effective data placement decisions in storage systems. Different from application hints, which can be interpreted by a storage system in different ways or simply ignored [41], semantic information in hStorage-DB requests a storage system to make data placement decisions. In particular, hStorage-DB has the following unique advantages.

First, rich semantic information in a DBMS can be effectively used for data placement decisions among diverse storage devices, as have been shown in our experiments. Existing approaches have employed various block-level detection and monitoring methods but cannot directly exploit such semantic information that is already available in a DBMS.

Second, some semantic information that plays an important role in storage management cannot be detected. Take temporary data as an example. The hStorage-DB performs effective storage management by (1) caching temporary data during its lifetime, and (2) immediately evicting temporary data out of cache at the end of its lifetime. In TAC [24], temporary data writes would directly go to the HDD, instead of the SSD cache, because such data are newly generated, and are not likely to have a "dirty" version in the cache that needs to be updated. In the three alternatives from [25], only DW and LC are able to cache generated temporary data. However, in the end of lifetime, temporary data cannot be immediately evicted out of cache to free up space for other useful data.

Furthermore, some semantic information, such as access patterns, may be detected, but with a considerable cost. Within hStorage-DB, such information is utilized with no extra overhead. For small queries, the execution may be finished before its access pattern could be identified. One example is related to sequential requests. In [25], special information from SQL Server is used to prevent special sequential request from being cached in SSD cache. In contrast, hStorage-DB attempts to systematically collect critical semantic information in a large scope.

8 Conclusion and Future Work

This paper presents the motivation, design, implementation, and evaluation of Hetero-DB, a new database system on heterogeneous computing and storage hardware. We redesign the query execution engine to well utilize GPU and multi-core CPU, and proposes a new storage engine by exploiting the hybrid HDD-SSD storage system. In the future, we will perform an overall evaluation on the system and continue optimizing Hetero-DB to adapt to the most advancement of new hardwares.

We will extend Hetero-DB in three directions. First, considering the importance of CPU-GPU coprocessing for high-performance query executions, we plan to investigate the probability of combining CPU scheduling in the operating system with the GPU scheduling in our system. Second, under the guideline of minimal changes to query engines, current query scheduler in Hetero-DB does not consider data overlapping between different queries. We will study how to coordinate the two layers to support data sharing. Third, we will also consider semantic information from database applications.

References

[1] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1021–1032. VLDB Endowment, 2004.

[2] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[3] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[4] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*, 2011.

[5] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[6] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[7] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. In *VLDB*, 2010.

[8] Evangelia A. Sitaridi and Kenneth A. Ross. Ameliorating memory contention of olap operators on gpu processors. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 39–47, New York, NY, USA, 2012. ACM.

[9] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, February 2011.

[10] B. He, M. Liu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems*, 34(4), December 2009.

[11] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62, New York, NY, USA, 2012. ACM.

[12] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 2011.

[13] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.

[14] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, 2008.

[15] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *Proc. VLDB Endow.*, 5(11):1543–1554, July 2012.

[16] J. Handy. Flash memory vs. hard disk drives - which will win? http://www.storagesearch.com/semico-art1.html.

[17] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1075–1086, New York, NY, USA, 2008. ACM.

[18] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45(1):45–53, February 2011.

[19] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. Gdm: Device memory management for gpgpu computing. *SIGMETRICS Perform. Eval. Rev.*, 42(1):533–545, June 2014.

[20] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for db2 using solid state storage. *Proc. VLDB Endow.*, 2(2):1318–1329, August 2009.

[21] Avinatan Hassidim. Cache replacement policies for multicore processors, 2010.

[22] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 26–26, Berkeley, CA, USA, 2004. USENIX Association.

[23] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. Ssd bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, September 2010.

[25] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM.

[26] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[27] C. Balkesen, J. Teubner, G. Alonso, and T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[28] S. Blanas, Y. Li, and J. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[29] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5), 2009.

[30] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, August 2013.

[32] Max Heimel and Volker Markl. A first step towards gpu-assisted query optimization, 2012.

[33] Sudhakar Yalamanchili. Second international workshop on performance analysis of workload optimized systems (fastpath-2013), held with ispass-2013. In *Scaling Data Warehousing Applications using GPUs*, April 2013.

[34] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 508–519, March 2014.

[35] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[36] Sebastian Breßand Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, August 2013.

[37] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[38] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[39] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the*

2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[40] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[41] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. Clic: Client-informed caching for storage servers. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 297–310, Berkeley, CA, USA, 2009. USENIX Association.