GHive: Accelerating Analytical Query Processing in Apache Hive via CPU-GPU Heterogeneous Computing

Haotian Liu[1], Bo Tang[1], Jiashu Zhang[1], Yangshen Deng[1], Xiao Yan[1], Xinying Zheng[1], Qiaomu Shen[1], Dan Zeng[1], Zunyao Mao[1], Chaozu Zhang[1], Zhengxin You[1], Zhihao Wang[1], Runzhe Jiang[1], Fang Wang[2], Man Lung Yiu[2], Huan Li[3], Mingji Han[4], Qian Li[1,5], Zhenghai Luo[5]
[1] Research Inst. of Trustworthy Autonomous Systems, Southern University of Science and Technology, Department of Computer Science and Engineering, Southern University of Science and Technology,
[2] The Hong Kong Polytechnic University, [3] Aalborg University,
[4] Boston University, [5] Huawei Technologies Co., Ltd.
Corresponding email: dbgroup@sustech.edu.cn

ABSTRACT

As a popular distributed data warehouse system, Apache Hive has been widely used for big data analytics in many organizations. Meanwhile, exploiting the massive parallelism of GPU to accelerate online analytical processing (OLAP) has been extensively explored in the database community. In this paper, we present GHive, which enhances CPU-based Hive via CPU-GPU heterogeneous computing. GHive is designed for the business intelligence applications and provides the same API as Hive for compatibility. To run SQL queries jointly on both CPU and GPU, GHive comes with three key techniques: (i) a novel data model gTable, which is column-based and enables efficient data movement between CPU memory and GPU memory; (ii) a GPU-based operator library Panda, which provides a complete set of SQL operators with extensively optimized GPU implementations; (iii) a hardware-aware MapReduce job placement scheme, which puts jobs judiciously on either GPU or CPU via a cost-based approach. In the experiments, we observe that GHive outperforms Hive in both query processing speed and operating expense on the Star Schema Benchmark (SSB).

ACM Reference Format:
H. Liu, B. Tang, et al.. 2022. GHive: Accelerating Analytical Query Processing in Apache Hive via CPU-GPU Heterogeneous Computing . In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3542929.3563503

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563503

1 INTRODUCTION

Open sourced by Facebook in 2008, Hive was designed to support distributed big data analytics on a massive scale [49, 50]. With a SQL-like interface on top of Hadoop MapReduce [3], Hive allows users to issue queries with familiar SQL semantics instead of implementing low-level MapReduce jobs. Hive has an active community, where developers provide enhancements in various aspects. For example, Yin et al. improve Hive's query processing performance by introducing optimized row columnar (ORC) file format, physical optimizations, and vectorized query execution [31]. Hortonworks Inc. renovates Hive along four axes, i.e., SQL and ACID support, optimization techniques, runtime latency, and federation capabilities [18]. Besides Hadoop MapReduce, Hive now also supports execution engines including Apache Tez [46] and Spark [4] by running on top of YARN (i.e., Hadoop's resource negotiator) [51]. Due to its stability and rich features, Hive was extensively utilized in production for large-scale data analytics by many organizations (e.g, Facebook, Huawei).

Many works in the database community explore using the massive parallelism of GPU to accelerate Online Analytical Processing (OLAP) [9, 12, 21, 22, 25, 28, 35, 38, 40, 43, 45, 47, 52, 57]. For example, GPUQP [25] considers query processing on a single machine and uses GPU as a co-processor for CPU. GPL pipelines the operators in a query and uses OpenCL to parallelize the execution of multiple GPU kernels. Based on LLVM (a compiler toolbox), HAPE [23], Red Fox [53] and HetExchange [21] build JIT-compilation frameworks to generate code for query execution on GPU. G-PICS [37] runs many spatial queries concurrently on GPU. These works achieve significant speedups compared to CPU-only execution, especially for computation intensive queries [47].

Given the popularity of Hive and the success of GPU-based OLAP acceleration, we started the GHive project with two main motivations. On the one hand, we observed that the computation intensive operators dominate the execution time for most of the queries in Hive. GPU can effectively

accelerate these operators [21, 22, 35, 47], and hence query execution time. On the other hand, our clusters have both CPU and GPU, and the GPU utilization has large fluctuations, leaving a large amount of transient GPU resources (as in Microsoft [32]). By enabling Hive to exploit GPUs, GHive can utilize these transient GPU resources and boost GPU utilization.

We set two main design goals for GHive. (1) Maintain the same interface and distributed execution management (i.e., via Yarn) as Hive, in order to simplify deployment and run many existing Hive applications without change. (2) Judiciously use CPU and GPU to reduce both query execution time and cost. Existing GPU databases cannot meet our design goals as they either do not support distributed query execution (e.g, GPUQP [25], GPUDB [55]) or conduct resource management with their own framework (e.g., Ominisci [12], BlazingSQL [5]). In addition, most of them assume that data are already in GPU memory while large-scale data usually reside on disk and can only be loaded to GPU in segments.

In GHive, we take the vectorized physical execution plan generated by Hive's query optimizer, which contains MapReduce jobs, and execute each job on either CPU (as in vanilla Hive) or GPU. Other functionalities, such as query optimization, distributed execution management, and data storage, are all handled by vanilla Hive for compatibility. There are three challenges in making GHive efficient. (1) MapReduce jobs are Java programs in vanilla Hive while current GPU libraries (e.g., CUDA and OpenCL) are in C++, which results in costly cross-language transferring for job execution on GPU. Moreover, the overhead of moving data to/from GPU memory can be substantial. (2) A complete set of efficient GPU-based SQL operators is required for job execution on GPU. (3) Each job needs to be placed judiciously on either GPU or CPU for good performance. To tackle these challenges, we design three key technical components in GHive.

- A compact data model: gTable. For efficient data movement from the address space of Java programs on the CPU (i.e., CPU executed jobs) to C++ programs on the GPU (i.e., GPU executed jobs), we design a column-based data model gTable. It achieves high efficiency by (i) converting the VectorizedRowBatch data model in Hive to gTable in CPU memory on-the-fly, and (ii) transferring only table columns that are necessary for job execution into GPU memory. (Section 4.1)
- GPU-based SQL operator library: Panda. Panda is general by supporting a complete set of common data types (e.g., int, float, double and string) and SQL operators (e.g., *hash join*, *sort merge join*, and *group by*). Panda is also efficient by implementing the operators using an indexing-based processing model, which reduces the data movement between CPU and GPU. (Section 4.2).
- Cost-aware job placement. For GHive to gain good performance, jobs should be placed on GPU only when the overhead of data movement can be outweighed by fast computation. To this end, we propose cost models to estimate job execution time on both CPU and GPU, and utilize GPU execution only when it is preferable. Our cost models are effective as they take both hardware characteristics and job properties into consideration. (Section 4.3)

We compared GHive with Hive under different hardware configurations and data scales on the famous SSB benchmark [42]. The results show that GHive outperforms vanilla Hive in both query processing speed and operating cost, in spite of the heavy overheads of language conversion and data movement imposed by industrial requirements. In particular, GHive achieves over 2X speedup over Hive for the computation-intensive queries. This is remarkable as the query execution time also includes disk I/O and network communication, which cannot be accelerated by GPU. When it comes to the execution time of individual operators, GHive often has orders of magnitude speedup over Hive. Moreover, we define the cost of running a query as the overall power consumption in its execution process and calculate the ratio of GHive cost over Hive cost. The results show that GHive consistently achieves lower processing cost for all queries in SSB. We also analyze the execution statistics of some representative queries to understand the performance of GHive and validate the effectiveness of our designs. The source code of GHive is available on GitHub [7].

The rest of this paper is organized as follows. Section 2 introduces how Hive processes a query as background and motivates GHive using an example query. Section 3 presents the overall architecture of GHive while Section 4 elaborates the three key technical components of it. Section 5 reports the experimental results. Section 6 reviews the related work, and Section 7 draws the concluding remarks.

2 BACKGROUND AND MOTIVATION

Query processing in Hive. As shown in Figure 1, Hive executes a SQL query by first translating it into a direct acyclic graph (DAG) of jobs and then submitting the jobs to Yarn for scheduling on the nodes in a cluster. Specifically, a SQL query is translated into a job DAG via four steps: (1) the *parser* parses the query to an *Abstract Syntax Tree (AST)*; (2) the *Calcite-based query optimizer* [16] translates the AST into an optimized logical execution plan; (3) the *converter* and the *physical optimizer* convert the logical plan into an vectorized physical execution plan; and (4) the underlying *execution framework* generates the job DAG based on the vectorized physical plan. There are two types of jobs in the DAG, i.e., Map jobs and Reducer jobs. Early versions of Hive use Hadoop MapReduce [24] as the execution framework,

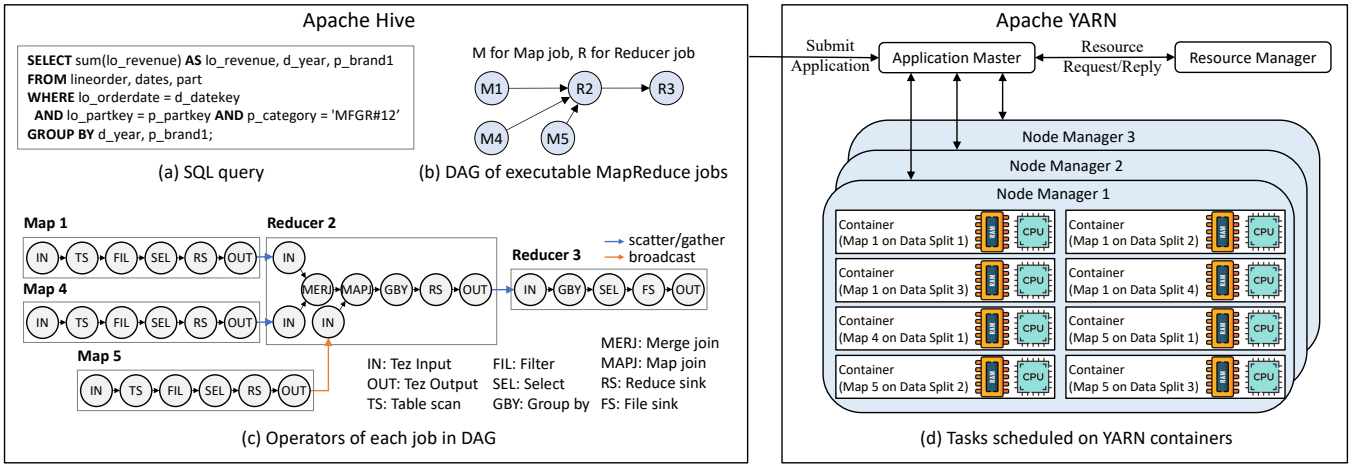


Figure 1: A SQL query, its directed acyclic graph of Map/Reduce jobs, and the tasks scheduled by YARN

which runs a Reducer job after each Map job and persist intermediate results on the disk. After version 0.13, Hive employs Apache Tez [46] as the execution framework, which reduces the overhead of data persistence. Thus, we use Tez as the CPU execution framework for GHive.

As shown in Figure 1(c), each MapReduce job contains a sequence of operators that correspond to a sub-query of the user-specified SQL query. For example, Reducer 2 involves MergeJoin, MapJoin, GroupBy, and ReduceSink. Tez further divides each job into three phases: (1) *input* (e.g., OrderedGroupedKVInput), which prepares input data for the job by loading from disk or upstream jobs in the DAG; (2) *processing*, which runs the operators of the job in a pipelined fashion; and (3) *output* (e.g., OrderedPartitionedKVOutput), which generates output data for the job. Tez conducts sorting when gathering/scattering data in the input/output phase such that the input data for MergeJoin and GroupBy are sorted. To execute a job, Tez starts multiple parallel tasks, each running the same sequence of operators specified by the job but on different partitions of the input data. As shown in Figure 1(d), Yarn schedules each task on a container for execution, and each container is a logical collection of physical resources (e.g., memory and CPU) on one node.

Opportunities and challenges for GPU acceleration. Figure 2 shows the execution process of the example query in Figure 1(a) on Hive with CPU. The query runs on the data of the SSB benchmark with scale factor 50 using our Cluster A, and the detailed experiment settings can be found in Section 5. The results show that the MapReduce jobs can be categorized into two classes, i.e., compute-bound and I/O-bound. For example, Reducer 2 is compute-bound as executing the operators takes up 80.3% of its running time. In contrast, Map 4 is I/O-bound as computation only contributes to 28.7% of its running time. Figures 2(a) and (b)

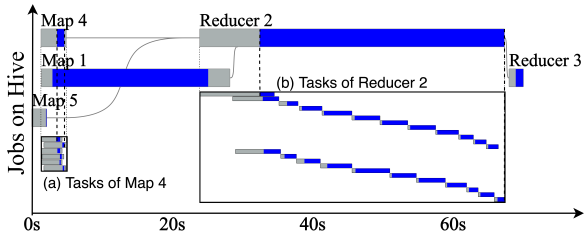


Figure 2: Execution process of a query on vanilla Hive, ***gray* indicates I/O time while *blue* indicates computation time, each zoom-in box shows the tasks for a job**

illustrate the execution process of the tasks of Map 4 and Reducer 2, respectively, which show that the tasks of the same job exhibit similar compute-I/O pattern. More importantly, Figure 2 also shows that the compute-bound jobs dominate the execution time of the query. This case is common for our queries as they usually contain computation intensive operators such as MergeJoin and GroupBy. Thus, if we can accelerate the compute-bound jobs using GPU, the running time of entire queries can be significantly reduced.

However, it is challenging to extend Hive to CPU-GPU heterogeneous computing for three reasons. (1) GPU execution introduces the overheads of language conversion (i.e., between JAVA and C++) and data movement (between CPU memory and GPU memory). (2) A GPU operator library that supports a complete set of data types and SQL operators is required such that existing Hive applications can run without change. (3) Only the compute-bound jobs should run on GPU as the I/O-bound jobs may run even slower. By tackling these challenges, we built the GHive system with our industry partner *Huawei* since January 2020 in order to accelerate analytical query processing for the business intelligence applications in its production environment.

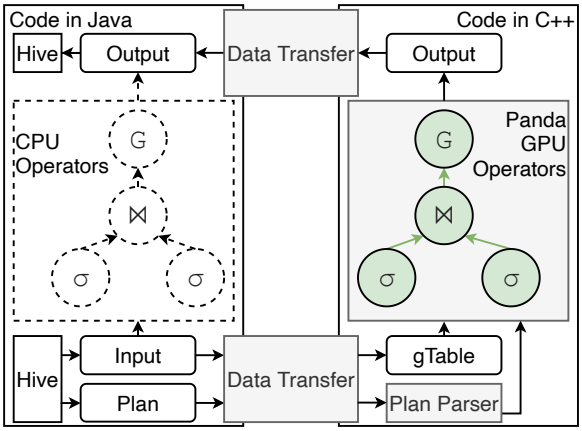
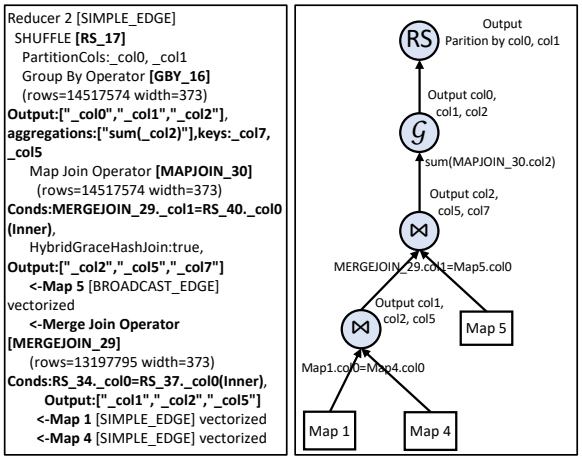


Figure 3: Job execution on GPU in GHive



(a) Plan text generated by Hive (b) C++-based execution plan tree

Figure 4: An illustration of job execution plan parsing

3 THE ARCHITECTURE OF GHIVE

GHive is designed to (i) maintain backward comparability with Hive such that existing Hive queries can run without change, and (ii) work with the Yarn-based resource management in our cluster. Specifically, GHive reuses the components of vanilla Hive to generate the vectorized physical plan (i.e., as discussed in Section 2) and converts the plan to a heterogeneity-aware job DAG, which executes some jobs on CPU and the other jobs on GPU. To this end, we first parse the vectorized physical plan with our plan parser (will be discussed shortly). Then, our hardware-aware scheduler (Section 4.3) will decide the job placement by analyzing its operators and involved data. The jobs assigned to CPU are executed by vanilla Hive. To run jobs efficiently on GPU, we propose a tailored data model gTable (Section 4.1) for cross-language and cross-device data movement, and a GPU-based SQL operator library Panda (Section 4.2).

Figure 3 shows how GHive executes (one task of) a job on GPU. Firstly, we move the input data and job execution plan generated by Hive from JVM to system memory, where our C++ code takes control. Secondly, we transform the input data from the data model in vanilla Hive (i.e., VectorizedRowBatch) to our gTable. The plan parser also parses the execution plan generated by Hive to C++-based plan for GPU execution. Thirdly, we execute the plan by incurring the corresponding GPU-based operators in Panda. Finally, the output data of the job are transferred back to JVM. This execution pattern ensures that a GPU executed job maintains the same input/output data as its CPU executed counterpart, and thus the jobs can take inputs from each other without considering the underlying hardware.

Job execution plan parser in GHive. Figure 4(a) shows an example of the plan text generated by Hive in the vectorized physical plan. The plan parer processes the plan text to (i) extract information about the operators and input data for a job such that the hardware-aware scheduler can make job placement decision, and (ii) determine how a job should be executed on GPU, which requires to translate the plan text into the C++-based execution plan tree in Figure 4(b).

To achieve the first goal, we extract all operators and their estimated result cardinalities (e.g., **MAPJOIN_30** with **rows=14517547**) from the plan text. To achieve the second goal, the parser takes five steps. (1) Extract all operators in the job (e.g., **MAPJOIN_30**) from the plan text. (2) Determine the topological order of the operators according to the indent. (3) Analyze the data dependency of each operator. Specifically, each input data of an operator comes either from another operator (e.g., **GBY_16** is the output of **MAPJOIN_30**) or another job (e.g., **MERGEJOIN_29** are from **Map 1** and **Map 4**). (4) Identify the conditions of each operator by tracking keywords such as **Conds**, **keys**, and **aggregations**. (5) Assign the output columns to each operator by parsing the lines beginning with **Output**. For example, the output columns of **MAPJOIN_30** are **_col2,_col5,_col7**.

4 KEY TECHNICAL COMPONENTS

In this section, we present the three key technical components of GHive. Specifically, we introduce the gTable data model and how to conduct data movement with it in Section 4.1, present GHive's SQL operator library Panda in Section 4.2, and discuss our cost-based scheduler that judiciously places each job on either CPU or GPU in Section 4.3.

4.1 Data Model: gTable

We first introduce the VectorizedRowBatch data model in Hive and discuss why it is not suitable for GPU execution and data movement, and then present our gTable data model.

Hive **data model: VectorizedRowBatch.** The VectorizedRowBatch model stores a batch of table rows (1024 by default) in a column-based layout to exploit the SIMD support

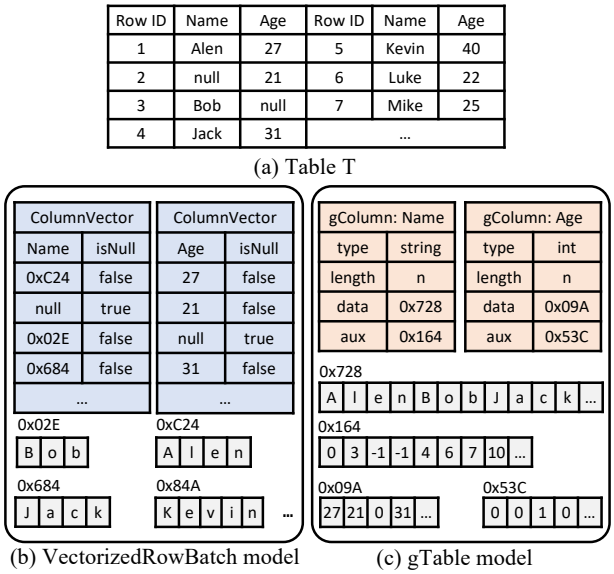


Figure 5: A table in the VectorizedRowBatch data model of Hive and the gTable data model of GHive

of modern CPUs [31], and each column is stored as a ColumnVector. Consider table T in Figure 5(a), its VectorizedRowBatch model representation is illustrated in Figure 5(b). For a string column (e.g., Name), VectorizedRowBatch stores each string as an array of bytes, and the strings may not be consecutive in memory. The ColumnVector of a string column stores the references to the strings, for example, the reference of Alen (i.e., 0xC24) is stored in the ColumnVector of Name. For a column of primitive data types, its ColumnVector directly stores the attribute values, see the ColumnVector of Age in Figure 5(b). Besides, ColumnVector also supports complex data types (e.g., DecimalColumnVector). To handle null values in the attributes, each ColumnVector in VectorizedRowBatch has a binary isNull array, which indicates whether the attribute value is null for each row.

VectorizedRowBatch is inefficient for job execution on GPU and data movement for two reasons. Firstly, each batch of table rows is small, and thus VectorizedRowBatch cannot fully utilize the massive parallelism of GPU. Secondly, for data types with variable length (e.g., string), VectorizedRowBatch incurs extra overheads for data movement and processing. On the one hand, transferring non-consecutive strings from CPU memory to GPU memory is inefficient as the PCIe bandwidth cannot be fully exploited. To exemplify, the names of 3rd and 4th rows (i.e., 'Bob' and 'Jack') of table T are located at two disjunct addresses 0x02E and 0x684. To move them to GPU, we need to invoke PCIe transfer twice and move a tiny amount of data each time. On the other hand, the GPU kernels have to locate the attribute value of each row during processing, which incurs extra overhead for the

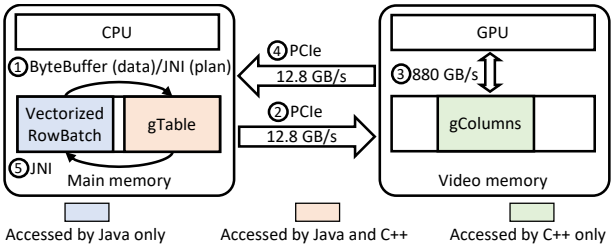


Figure 6: CPU-GPU data movement in GHive

variable-length data types. One may encode the variable-length data into integers [17, 26, 29, 41, 47] but the cost of such encoding is high [27].

GHive data model: gTable. To tackle the limitations of VectorizedRowBatch, we propose the gTable data model, which also adopts a columnar-based layout as shown in Figure 5(c). Specifically, gTable stores each column as a gColumn, and each gColumn has four meta-attributes: column data type (type), the number of rows (length), the reference of the array of data values (data), and the reference of an auxiliary information array (aux). For a column with variable-length data types, gColumn concatenates all attribute values such that they are consecutive in memory and uses the auxiliary array to locate the attribute value for each row by indirect addressing. Consider the gColumn:Name in Figure 5(c), the reference of the array of data values is 0x728, which locates the starting address of the strings for column Name in table T. The aux array of gColumn:Name stores the starting and ending positions of each string in the data array, and both positions are set to -1 when the corresponding string is null, like the 2nd row for column Name. For a column with primitive data types (e.g., gColumn:Age), gColumn stores the attribute values in the data array, and uses a bitmap as the aux array to indicate the null rows, which is similar to the isNull array in VectorizedRowBatch. gTable ensures that data of a column are always consecutive in memory, and thus facilitates efficient data movement and processing.

Data movement in GHive. Figure 6 shows how the gTable data model is used for data movement between CPU and GPU. Specifically, the first step of data movement (see step 1 in Figure 6) is transferring both data and Java-based execution plan into an address space on CPU that can be accessed by C++ programs. In particular, step 1 involves two tasks: (i) transforming the data from VectorizedRowBatch to gTable model and (ii) parsing the operator execution plan (in Java) to GPU-based execution plan (in C++). Then the data are moved to the video memory of GPU from CPU memory via PCIe bus in step 2. The GPU executes the instructions of the GPU-based operators on data in the gTable model at step 3. The output data of the job are moved from GPU

to the address space of C++ program at step 4. Finally, the output data are transferred to the address space of JVM and transformed into the VectorizedRowBatch model at step 5. Steps 2, 3, and 4 are standard for applications that use GPU as a co-processor of CPU [54], and step 5 is straightforward. Thus, we introduce how GHive transforms data from the VectorizedRowBatch model to the gTable model as follows.

In Hive, the input data of MapReduce jobs come row by row, or as VectorizedRowBatch objects. The MapRecordSource or ReduceRecordSource operator of each job collects input data and passes them to downstream operators of the job for execution. In GHive, we revise MapRecordSource and ReduceRecordSource to collect data into a ByteBuffer, which reside in a shared memory that can be accessed by both Java and C++. ByteBuffers are used to store the data values and auxiliary information of each column, and each row is added to the ByteBuffers on-the-fly when it comes. We initialize the size of each ByteBuffer as 16MB and create a new ByteBuffer with twice the capacity of the original ByteBuffer when a ByteBuffer is full. All data in the original ByteBuffer are copied to the new ByteBuffer. Figure 7(a) reports the data collection time when using different initial sizes for the ByteBuffer. The results show that larger initial size leads to shorter collection time but the performance gain is not significant. Thus, we set the initial buffer size as 16MB to avoid using too much memory for categorical columns. After collecting all input data, the ByteBuffers are used to construct the gTable model. Note that we copy the references of the data value and auxiliary information arrays instead of the actual data from the ByteBuffers to gTable, which reduces time and memory comsuption. We also control the input data size of the jobs by configuring the bytesPerReducer parameter in Hive such that the ByteBuffer (resp. gTable) does not cause OOM on CPU (resp. GPU).

Alternative solutions. One alternative to our ByteBuffer is passing data to GPU as parameters using JNI. We compare JNI and ByteBuffer on one machine in our Cluster A (see Section 5 for detailed experiment settings). The results in Figure 7(b) show that ByteBuffer performs significantly better than JNI. This is because ByteBuffer only conducts data copy once while JNI conducts data copy twice. Moreover, the gain of ByteBuffer over JNI is moderate when data size is small but becomes significant when data size increases. Thus, we use JNI to move the execution plan because the execution plan is small in size and using JNI is easy via standard function call. Another alternative is Apache Arrow [1], which allows to migrate data among different systems by defining a unified format. We do not use Apache Arrow as integrating it into Hive takes substantial engineering effort and its general design introduces extra overhead. As shown in Figure 7(b), Apache Arrow is even slower than JNI.

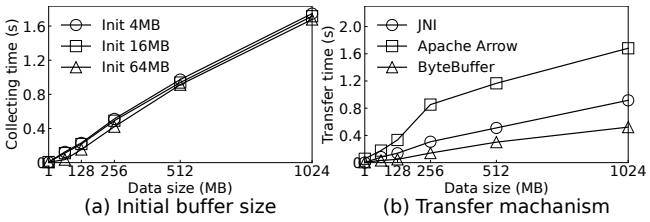


Figure 7: Design choices for data movement

4.2 GPU-based Operator Library: Panda

Many works studied accelerating SQL queries with GPU-based operator implementations [9, 25, 47, 48], and we will discuss them in Section 6. We decide to develop our own GPU-based operator library Panda for GHive as existing operator implementations do not consider the execution pattern of GHive and are limited in their support for data types and operators. As moving data between CPU and GPU is a major overhead, Panda adopts an indexing-based processing model to reduce data movement for efficiency. Panda also achieves generality by supporting a wide range of data types and SQL operators with sensible implementations.

Indexing-based processing model. Existing GPU-based SQL operator implementations assume that the relational tables reside in GPU memory as a whole. This is inefficient for GHive as GHive needs to move data from CPU memory to GPU memory for execution, and usually only some (instead of all) columns of a table are utilized to run an operator. Thus, the overhead of moving the irrelevant columns to GPU are unnecessary. Using the idea of late materialization [15, 45], we design an indexing-based processing model to avoid transferring unnecessary data. In particular, our indexing-based processing model works as follows. (1) It assigns an unique id (denoted as idx) for each row in a table and associates the ids with each gColumn. (2) To run each operator on GPU, it only moves the relevant columns to GPU memory (e.g., filter or join on some columns). (3) When computing the results of an operator, the ids of the rows are kept such that results can be reconstructed from the original tables. Note that our gTable naturally supports the indexing-based processing model as it stores each column separately as a gColumn.

Figure 8 shows an example of the indexing-based processing model for the simple query in the lower-right corner. We only move gColumn:datekey of table A and gColumn:orderdate of table B into GPU memory, as only these two columns will be used by the join operator in GPU. Then, we run the join operator on the two columns and obtain the indexing-based results. The indexing-based results are moved back to CPU memory and used to fetch the relevant rows in the relational tables for the final results.

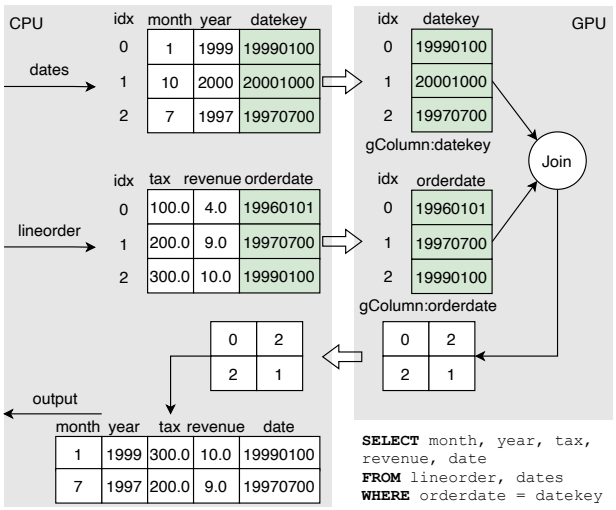


Figure 8: Indexing-based processing model example

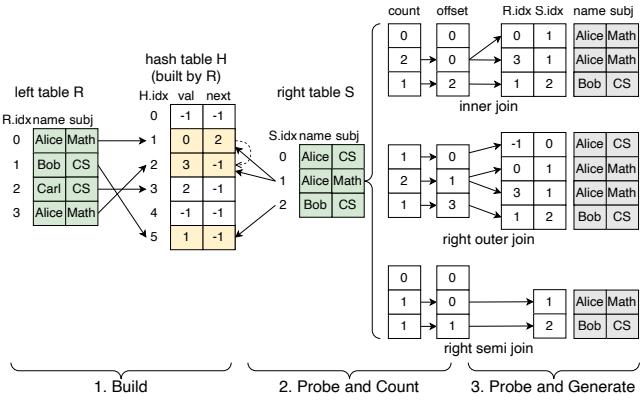


Figure 9: Example of hash join implementation

To execute a job, the indexing-based processing model moves data to GPU one operator at a time (instead of for all operators in a batch before execution) to exploit the filtering ability of the indices. For example, a job may filter gColumn:datekey of table A, and then join gColumn:year of table A's filtering results with a column from another table. In this case, the indexing-based results of the filter operator is used to fetch only the relevant rows in gColumn:year for the join operator. We observed that the indexing-based data loading can significantly reduce data movement for some operators. If a gColumn is already in GPU memory, we do not construct the exact result according to the indexing-based results immediately after executing an operator. Instead, we process the downstream operators by indirectly addressing the gColumn using the indexing-based results. In particular, we use the fancy iterator *thrust::permutation_iterator* for efficient indirect addressing.

To sum up, our indexing-based processing model moves only the necessary columns to GPU and executes the GPU operators by indirect addressing. It reduces data transfer via PCIe and consumption of GPU memory. Besides, it helps to make the GPU operators general as we will show using the example of hash join. Similar ideas of late materialization (by recording row indexes) are also used by column-based databases like Vertica and MonetDB but for different purposes, e.g., reducing intermediate result construction, executing on compressed data, improving cache performance and supporting vectorized optimizations on CPU.

Type and operator support. Comprehensive data type and operator support is essential for GHive as we have many existing Hive queries that cover most data types and operators in OLAP. To this end, we build Panda from scratch using NVIDIA cub [6] and thrust [14] libraries, which are widely used in industry due to their flexibility and generality. Currently, Panda supports all common data types (e.g., int, float, double, string), and we are working to integrate complex data types such as *DATE* and *TIMESTAMP*. We use hash join as an example to show the generality of Panda. With sensible implementations, our hash join can handle multiple join keys and different types of join (i.e., inner, outer or semi).

Figure 9 shows how GPU-based hash join is implemented in Panda. Both tables R and S have three columns, and we use the name and subj columns as join keys to illustrate how Panda supports multi-key join. Note that column idx is the row index, which is added by our indexing-based processing model. Hash join is conducted in two phases, i.e., (i) *hash table building*, and (ii) *hash table probing*. The tuples in table R are used to populate the hash table in the building phase, while each tuple in table S (i.e., probe table) finds its relevant rows from table R in the hash table during the probing phase.

As shown in Figure 9, R's hash table has two columns. In each row, the 1st column stores the index of the row in R that hashes to the bucket (-1 if the hash bucket is empty), the 2nd column stores the index of the next entry in the hash table that has the same hash value as the row in the 1st column (used to handle hash collision). Consider the hash table in Figure 9, its 2nd row is $\langle 0, 2\rangle$, where the first value '0' means that this hash bucket contains row $R[0]$ in table R, i.e., $\langle 0, Alice, Math\rangle$. The second value '2' indicates the next hash bucket id that has the same hash value as the join key of $R[0]$ (i.e., $H[2]$). We have $R[3]$ in the first column of $H[2]$ as the join key of both $R[0]$ and $R[3]$ are $\langle Alice, Math\rangle$. The GPU threads work in parallel to build the hash table, and each thread processes a subset of the rows in table R. We use CUDA atomicCAS instruction to update the hash table in order to avoid the errors caused by multiple threads accessing the same hash bucket.

Table 1: Customizable functions for Panda operators

Operator	Extendable Interface
Hash join	hashing, probing
Group by	sorting, comparator, aggregator
Filter	filter condition
Select	user-defined function

The probing phase contains two stages (i) *probe and count*, and (ii) *probe and generate*, which are separated by a barrier. In stage (i), we initialize two integer arrays to store the Count and Offset for each row of table S, as shown in Figure 9. Every tuple in S probes the hash table and counts the number of matched rows in table R in parallel, and the results are stored in the Count array. Take inner hash join for an example, row $S[1]$ is $\langle 1, Alice, Math\rangle$, which finds 2 matched rows in R by probing the 2nd and 3rd hash buckets (i.e., $R[0]$ and $R[3]$). Thus, it fills 2 into Count[1]. Then the BlockScan primitive in the NVIDIA cub library is invoked to compute the prefix sum for the Count array as the Offset array, which tells where to store the join result for each row of table S. In stage (ii), we generate join results for the rows of S in parallel by storing the results in the output table according to the Offset. For example, in inner join, row $S[2]$ $\langle 2, Bob, CS\rangle$ generates a result tuple $\langle 1, 2\rangle$ with offset 2 in the output table.

We store the row index instead of the actual tuples when building the hash table as it allows to support multi-key join and different data types with one implementation. For example, to support three-key join or a new data type, only the hash function and equality checking function need to be updated without affecting the core join implementation. Our join implementation also supports different join methods, e.g., inner join, outer join and semi join. As shown in Figure 9, only the probing stage needs to be changed for different join methods while the three-phase join logic remains the same.

Extendability of Panda. Panda is highly extendable with simple interfaces for implementing new functions and operators. Specifically, Panda adopts the functor design pattern and leaves interfaces to customize significant functions for the operators, for which some examples are shown in Table 1. To implement an extended operator, engineers only need to inherit the parent *Operator* class and customize its *execute()* function. The references of the input and output data for each operator are defined in the *Operator* class, and engineers can directly read and write them in their own *execute()* function. Engineers can easily change the behavior of an operator by modifying its key functions. The code below shows an example of customizing the *comparator* functor for the sort-based *GroupBy* operator, with which GroupBy groups tuples whose keys are considered the same. The *operator()* function determines whether two keys are the same.

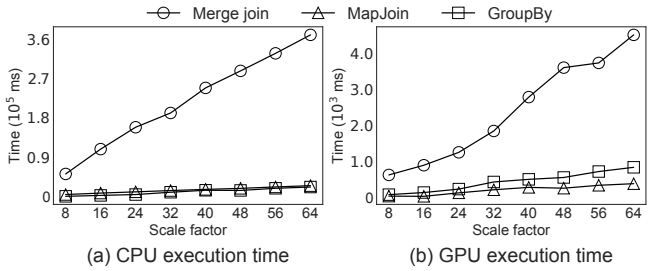


Figure 10: Operator execution time w.r.t scale factor

The functor design simplifies the procedure of supporting complex operations and new data types. It also allows to reuse well-optimized GPU-based operator implementations if the core operator logic is not changed.

```
struct abs_comparator{
    int *keys; // the int column of the group key
    __host__ __device__ // i, j are indexes
    bool operator()(const int i, const int j) {
        return abs(keys[i]) == abs(keys[j]);
    }
}
```

4.3 Hardware-aware Job Placement

For a query, GHive needs to decide each job should be executed on CPU or GPU. Although GPU execution enjoys fast computation due to massive parallelism, it also incurs the overhead of moving data between CPU memory and GPU memory. As we have shown in Figure 2, the MapReduce jobs can be classified into compute-bound and I/O-bound. For the I/O-bound jobs, GPU execution may be slower than CPU execution as the benefit of fast computation may not outweigh the data movement overhead. To judiciously place each job on the right device, we devise cost models by jointly considering the data, job and hardware.

Denote the execution time of a job J on CPU and GPU as $T_C(J)$ and $T_G(J)$, respectively. The job placement strategy of GHive is simple—*execute job J on GPU only if J runs significantly faster on GPU than on CPU*. Formally, GHive places job J on GPU when

$$\frac{T_C(J) - T_G(J)}{T_C(J)} \geq \theta,$$

where θ is set to 0.2 by default (meaning that the GPU execution time needs to be less than 80% of the CPU execution time) and tunable by users. Thus, the key is to estimate $T_C(J)$ and $T_G(J)$ accurately.

Cost model for CPU execution. For the CPU execution time $T_C(J)$, we only consider the operator processing time. This is because other time comsuptions, e.g., loading data from disk and sending data over network, are the same for

GPU execution. As introduced in Section 2, a job J consists of a sequence of operators, and thus we model $T_C(J)$ as

$$T_C(J) = \sum_{\forall op_i \in J} f(op_i, n_i), \tag{1}$$

where op_i is an operator in job J, $f(op_i, n_i)$ is the execution time of operator op_i, and n_i is the result cardinality of op_i. Thus, the problem becomes modeling $f(op_i, n_i)$. Figure 10(a) reports the execution time of some operators on CPU with different scale factors. We use the example query in Figure 1(a) and data from the SSB benchmark, and the experiment is conducted on our Cluster B (see the details in Section 5). The results show that the execution time of the operators are almost linear w.r.t data size. This is because linear cost model matches Hive's execution pattern. In particular, Hive uses MapJoin and MergeJoin to join the tables; the building and probing phases of MapJoin and merge phase of MergeJoin have linear complexity. Hive executes sorting by first processing each segment and then merging the segments, which also yields approximately linear complexity. We profiled other operators and many queries, and found that the linear trend is consistent.

Thus, we estimate the CPU execution time of each operator using standard linear regression, i.e., $f(op, n_i) = k_{op} n_i + b_{op}$, where k_{op} and b_{op} are the coefficient and intercept, respectively. Before handling queries, we obtain k_{op} and b_{op} by fitting the result cardinality and operator execution time measured in trials. Note that different operators use different models with separate parameters. To estimate the execution time for an operator, we use its operator type op_i and estimated result cardinality n_i (given by the query optimizer) to invoke the linear models.

Cost model for GPU execution. The cost of GPU-based job execution in GHive has three components: (1) moving data from CPU to GPU, (2) executing the operators on GPU, and (3) reconstruct the output data of the job for our indexing-based processing model and convert the data back to the VectorizedRowBatch data model. Thus, we model the GPU execution time as

$$T_G(J) = T_{pre}(J) + T_{exec}(J) + T_{post}(J). \tag{2}$$

(1) T_{pre} is the time to move input data from the address space of Java programs on CPU to GPU memory.
(2) T_{exec} is the time to execute the GPU-based operators.
(3) T_{post} is the time to generate output data for the job.

We model the three components separately as follows.

Estimating T_{pre}: The data preparation time T_{pre} includes step 1 (transforming data from VectorizedRowBatch model to gTable model) and step 2 (moving data from CPU memory to GPU memory) in Figure 6. Step 1 copies data in CPU memory and step 2 transfers consecutive data via PCIe bus,

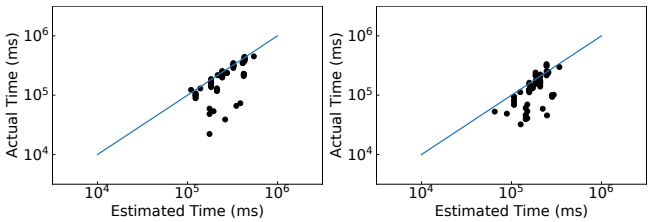


Figure 11: Comparison of operator execution time

and thus the costs of both steps are linear w.r.t data size. Thus, we model T_{pre} as

$$T_{pre}(J) = \frac{\sum_{\forall c_j \in C(J)} h(c_i)}{\alpha} + \frac{\sum_{\forall c_j \in C'(J)} h(c_i)}{\beta}, \tag{3}$$

where α and β are bandwidth of main memory and PCIe bus, respectively, and we obtain them via micro benchmark experiments. c_j is a column in the input tables for the job, set $C(J)$ contains all the columns while $C'(J)$ includes only the columns that are moved to GPU memory. $h(c_j)$ is the data size for column c_j.

Estimating $T_{exec}(J)$: We model $T_{exec}(J)$ as

$$T_{exec}(J) = \sum_{\forall op_i \in J} g(op_i, n_i), \tag{4}$$

where $g(op_i, n_i)$ is the GPU execution time of operator op_i and n_i is the result cardinality. The same as CPU-based operator execution, we use a linear model to estimate $g(op_i, n_i)$. In Figure 10(b), we profile the GPU execution time of some operators by using the same experiment configurations as Figure 10(a). The results show that the execution time has a linear trend w.r.t data size, and thus using a linear model is reasonable.

Estimating T_{post}: After executing a job on GPU, GHive needs an extra step to obtain the result table due to our indexing-based processing model. Specifically, we iterate over the indexing-based results and fetch the required rows from the data tables. Thus, we model T_{post} as

$$T_{post} = \gamma \cdot |R|, \tag{5}$$

where $|R|$ is the cardinality of result table and γ is the time to fetch a result row and concatenate it to the result table. To estimate T_{post}, $|R|$ is obtained from the cardinality estimations provided by the query optimizer, and γ is measured by micro benchmark experiments (for different data sizes).

To validate the accuracy of our cost models, in Figure 11, we report the estimated and actual execution time of some jobs sampled from SSB. These jobs vary in the scale of involved data and complexity (from zero to three joins). Note that Hive jobs are usually not complex as a query is decomposed into multiple jobs for execution and each job typically

Table 2: CPU and GPU information of two clusters

Hardware	Cluster A	Cluster B
CPU	Intel Xeon E5-2640 v4	Intel Xeon Gold 5122
CPU number	2	2
Core number	40	16
CPU memory	64GB	512GB
GPU	NVIDIA Tesla T4	NVIDIA TITAN Xp
GPU memory	16GB	12GB

contains only several operators. Figure 11 shows that the points scatter around the diagonal line, indicating good prediction accuracy. In particular, defining the relative error $\epsilon = \frac{|T_{pred} - T_{exact}|}{T_{exact}}$, the median errors of our cost models for CPU and GPU are 15.5% and 25.8%, respectively. We find that large errors in cost estimation are caused by large errors in the cardinality estimations of intermediate results, and we use the cardinality estimations provided by Hive's query optimizer. This echos the observation that linear cost models work well if the cardinality estimations are accurate [36], and we note that improving cardinality estimation is beyond the scope of this paper.

5 EXPERIMENTAL EVALUATION

In this section, we compare Hive and GHive for both query processing speed and operating cost. Specifically, Section 5.1 conducts overall performance evaluation using different cluster configurations, queries and data scales. Section 5.2 analyzes the execution process of two representative queries on Hive and GHive to verify the benefits of GPU execution and the effectiveness of our designs.

Experiment settings: We used the SSB benchmark [42] for our experiments, which contains 5 tables organized in star schema and is widely used in related researches [21, 45, 47, 57]. There are 13 queries in SSB and they involve most of the commonly used SQL operators such as selection, projection, join, sorting and aggregation. By default, we set the scale factor of the data to 50. To test the generality of GHive on different hardware platforms, we conducted the experiments on two clusters, each consisting of 4 homogeneous nodes.

In both clusters, the nodes are connected via 10 Gbps Ethernet, and the CPU and GPU on the same node are connected via PCIe 3.0 with x16 bandwidth. The CPU and GPU information of the two clusters is shown in Table 2. The version of NVIDIA driver, CUDA, Hadoop, Hive and Tez are 460.32.03, 11.2, 3.2.1, 3.1.0 and 0.10.1 respectively. To model the disk I/O overhead in big data analytics, we stored the tables in ORC format on HDFS when conducting the experiments.

5.1 Overall Performance Evaluation

Table 3 and Table 4 report the query execution time of Hive and GHive on Cluster A and Cluster B, respectively. We also include the *GPU utilization* of GHive in the query execution process, and the speedup of GHive over Hive. The *cost ratio* is the overall power consumed by GHive for query execution compared to that of Hive, and a value smaller than 1 indicates that GHive has a smaller operating cost than Hive. Specifically, the cost ratio is defined as

$$\mu = \frac{T_{\text{GHive}} \cdot (P^{CPU}_{\text{GHive}} + P^{GPU}_{\text{GHive}})}{T_{\text{Hive}} \cdot P^{CPU}_{\text{Hive}}}, \tag{6}$$

where T_{GHive} (*resp.* P^{CPU}_{GHive}) and T_{Hive} (*resp.* P^{CPU}_{Hive}) are the query execution time (*resp.* average CPU power in the query execution period) for GHive and Hive. P^{GPU}_{GHive} is the average GPU power for GHive. We measured CPU power using the RAPL tool [30] provided by Intel, and GPU power using the NVML library of NVIDIA [11]. We use power consumption as the operating expense because the machines are deployed and power bills dominate the costs for us as a cloud provider.

Table 3 and Table 4 show that GHive outperforms Hive for all queries in terms of both query processing time and operating cost. In addition, the performance advantage of GHive is consistent for the two clusters. The speedup can be over 2x and is large for all queries except for Q1.1, Q1.2, and Q1.3. This is because these queries contain few computation-intensive operators, and thus the gain of GPU execution is not significant. In contrast, Q4.1, Q4.2, and Q4.3 are more complex (having four joins, one aggregation, and one sorting), and thus the speedups of GHive are close to 2x. We note that the performance gain of GHive is remarkable as some important overheads, such as disk I/O and network communication cannot be reduced with GPU execution. As we will show in the case studies in Section 5.2, for the computation time of individual operators jobs, GHive can speedup Hive by orders of magnitude.

The results also show that GHive consistently achieves lower operating cost than Hive, and the maximum cost reduction can be over 25%. This is appealing for big organizations like Huawei as we have many routine Hive applications and are paying huge electricity bills for large clusters. As reported in the tables, the low operating cost of GHive is attributed to two reasons, i.e., shorter CPU running time by using GPU for acceleration and low GPU utilization. Note that the GPU utilization in Table 3 and Table 4 are overestimated as we regard a GPU as entirely occupied by GHive even if the actual device utilization rate is usually low. When considering actual device utilization[1], the GPU utilization of GHive is below 5% for all queries. The low GPU utilization of GHive

[1]In this case, GPU utilization is calculated by integrating actual device utilization over time.

Table 3: Performance comparison between Hive and GHive on Cluster A

Queries	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Hive time (s)	16.57	13.15	14.71	95.81	83.15	70.98	96.48	88.23	76.25	25.90	124.08	108.19	110.59
GHive time (s)	16.08	13.03	14.10	48.28	41.68	40.60	67.87	45.01	43.62	17.57	64.62	60.24	64.24
GPU utilization	7.02%	7.41%	13.28%	13.26%	14.88%	14.29%	5.50%	13.33%	14.21%	23.34%	9.90%	15.27%	15.10%
Speedup over Hive	**1.03**	**1.01**	**1.04**	**1.98**	**2.00**	**1.75**	**1.42**	**1.96**	**1.75**	**1.47**	**1.92**	**1.80**	**1.72**
Cost ratio (μ)	0.93	1.00	0.95	0.73	0.74	0.78	0.88	0.72	0.83	0.96	0.77	0.80	0.86

Table 4: Performance comparison between Hive and GHive on Cluster B

Queries	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Hive time (s)	15.33	15.48	12.33	97.54	87.62	79.71	97.25	90.75	83.68	22.05	124.54	109.84	109.04
GHive time (s)	14.30	14.72	11.83	43.57	45.59	41.83	59.90	40.76	41.17	16.17	60.47	59.98	63.05
GPU utilization	20.92%	18.62%	14.09%	17.67%	11.41%	12.91%	2.65%	14.48%	14.33%	24.74%	11.01%	11.17%	13.01%
Speedup over Hive	**1.07**	**1.05**	**1.04**	**2.24**	**1.92**	**1.91**	**1.62**	**2.23**	**2.03**	**1.36**	**2.06**	**1.83**	**1.73**
Cost ratio (μ)	0.91	0.94	0.92	0.77	0.78	0.77	0.91	0.74	0.78	0.94	0.76	0.86	0.85

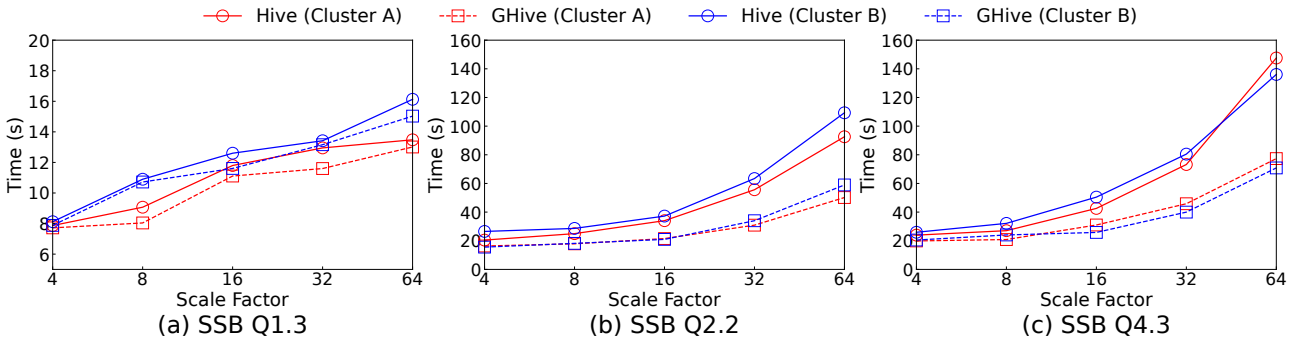


Figure 12: Performance comparison between Hive and GHive under different scale factors

makes it easy to share GPU resource with other applications to boost resource utilization using tools such as MPS [10].

Results under different data scales: Figure 12 compares the query processing time of Hive and GHive by varying the scale factor of SSB from 2 to 64. We choose 3 representative queries, i.e., Q1.3 for simple queries, Q2.2 and Q4.3 for complex queries. The results show that GHive (dashed lines) always outperforms Hive (solid lines) under all scale factors on both Cluster A (red) and Cluster B (blue). However, query processing time increases slowly with scale factor for Q1.3 but much more rapidly for Q2.2 and Q4.3. This is because Q1.3 has only one join operator and it is optimized as a relatively simple mapjoin[2] by Hive. In contrast, Q2.2 and Q4.3 has 3 and 4 join operators, respectively, and involve more complex merge join. Figure 12 also shows that for more complex queries, the performance gain of GHive over Hive is large and increases with data sizes. This is favorable for us as we have many complex queries over large-scale data.

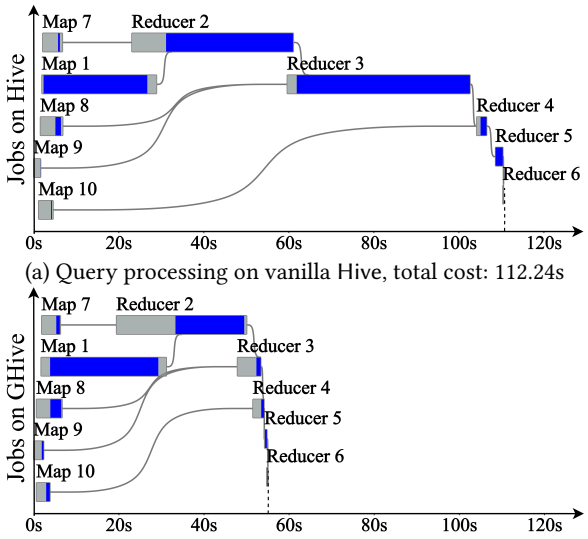


Figure 13: SSB Q4.3, scale factor 50

5.2 Case Study

To understand the superior performance of GHive over Hive, we analyze two queries as case study. One query is the Q4.3

[2]Mapjoin joins a big table with a small one by hashing the small table, and thus each row in the big table only needs to check a few entries

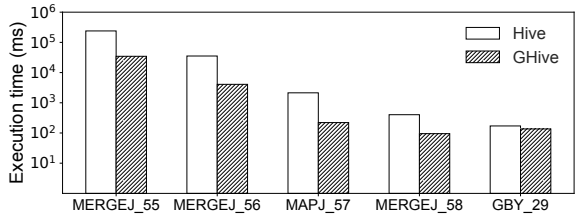
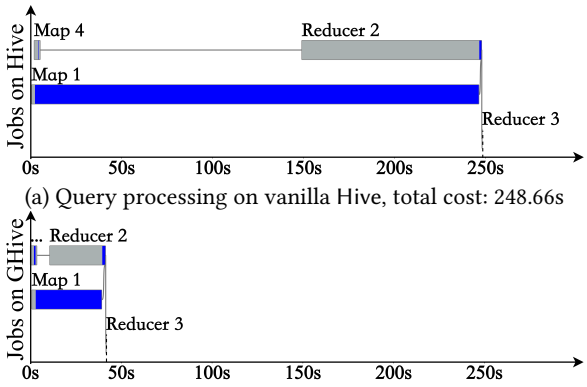


Figure 14: Comparison of operator execution time



(a) Query processing on vanilla Hive, total cost: 248.66s

(b) Query processing on GHive, total cost: 42.56s

Figure 15: Production query with decryption

of SSB and the other is the query of production. The results are obtained on Cluster B.

Figure 13 shows the query processing of Q4.3 on Hive and GHive, where each bar corresponds to a job and the x-axis represents time. For each job, the gray part indicates the time for data preparation (e.g., disk I/O, network) while the blue part indicates the time for computation (involving moving data to/from GPU in the case of GHive). The results show that Hive and GHive have the same job DAG, which consists of 5 map jobs and 5 reducer jobs. This is because GHive uses the vectorized physical plan generated by Hive. However, Q4.3 takes 111.24 seconds on Hive but only 55.80 seconds on GHive, yielding a 2x speedup. Closer inspection finds that this is because GHive significantly reduces the running time of Reducer 2 and Reducer 3 by placing them on GPU, and the two jobs dominate the query execution time for Hive. For Reducer 3, the computation time is long on Hive but almost negligible on GHive.

In Figure 14, we profile the execution time of the most computation-intensive operators in Q4.3 on Hive and GHive. The results show that on the operator level, GPU execution provides orders of magnitude speedup, which verifies the efficiency of our GPU-based operator library Panda. Note that in our profiling, we count the time for upstream sorting towards the execution time for MergeJoin and sort-based GroupBy. If a job is to be executed on GPU, we disable upstream sorting on CPU to save computation as our GPU operators do not require sorted input.

Figure 15 shows the execution process of the production query containing an UDF that conducts decryption. Note that we use encryption to protect sensitive information, and can only be accessed by authorized users with the private key. Figure 15(a) shows that Hive takes a long time to execute Map 1, which conducts decryption. This is because the decryption process involves heavy arithmetic operations and is compute intensive. Figure 15(b) shows that it can be significantly accelerated by GPUs. As a result, GHive speedups Hive by 5.8x for the production query.

6 RELATED WORK

In this section, we discuss relevant works on enhancements for Apache Hive and GPU-accelerated data management.

Apache Hive. There are many works that enhance Hive in different aspects. For example, YSmart [34] optimizes the translation from SQL query to MapReduce jobs by considering the correlation among the operators. Huai et al. [31] summarize major improvements to Hive from 2010 to 2014, which includes a new file format named optimized record columnar file (OCR file), advanced query optimization techniques, and vectorized query execution model. Calcite [16], an optimizer that supports both rule-based optimization (RBO) and cost-based optimization (CBO), is used by Hive for logical execution plan optimization. Hortonworks Inc improves Hive in transaction processing, query optimizer, runtime, and the ability to federate across different systems. For the execution engine, Hive moves from Hadoop MapReduce [24] to Apache Tez [46], which has smaller data persistence overhead and is more flexible in implementing applications. Our GHive extends Hive by supporting CPU-GPU heterogeneous computation. This is important as many organizations have GPUs in their clusters and GHive not only accelerates complex queries but also helps to fully utilize GPU resources.

GPU-accelerated data management. Some works study GPU-based SQL operator implementations. Paul et al. [44] survey works on GPU hash join and conduct a comprehensive performance evaluation for them. Sioulas et al. [48] propose a family of partitioning-based join algorithms that consider the limited memory capacity and slow PCIe bandwidth of GPU. Karnagel et al. [33] compare and analyze sort-based and hash-based GPU implementations for grouping and aggregation operators, and design a heuristic optimizer for the algorithms and parameters. By integrating different operators, several GPU-based SQL operator libraries are developed. Crystal provides a set of well-optimized primitives for implementing SQL operators on GPU [47]. However, substantial efforts are required to implement SQL operators using Crystal, for example, handling hash collisions and supporting string data type. cuDF [9] is a GPU DataFrame library that supports loading, joining, aggregating, and filtering data. It provides

GPU-based implementations for common SQL operators, and is used by RAPIDS[13]. Many implementation details of the operators in our Panda library are motivated by existing works. However, Panda comes with an indexing-based processing model to reduce the data movement between CPU and GPU. Moreover, we set generality and extendibility as first-class objectives when developing Panda, which makes it possible to integrate Panda into other big data systems such as Spark and Flink [2].

Many systems uses GPU as a co-processor for CPU in SQL query processing. GPUQP [25] targets a single machine and designs a cost model to place each operator on either CPU or GPU by considering computation and PCIe data transfer. Ocelot [28] is a hardware-oblivious extension of MonetDB that can run on different computation devices, which also supports GPU-based query processing. HetExchange [22] designs a framework to generate code for heterogeneous CPU-GPU platform using just-in-time (JIT) compilation and supports query execution with multiple CPUs and GPUs. As a GPU-based database system, OmniSciDB [12] keeps hot data in GPU memory for fast access during query execution. Our GHive differs from these works as it considers CPU-GPU heterogeneous computing for a distributed system.

Attempts have also been made to integrate GPU into big data systems [8, 19, 20, 56]. For example, FlinkCL [19] integrates GPU into FLink using JIT and provides a Java interface for writing GPU codes. There are also works exploiting other hardware characteristics to improve performance. For instance, HippogriffDB [39] directly transfers data from NVMe SSD to GPU and designs a GPU-friendly data compression mechanism to reduce data transfer. Our GHive is a systematic solution that accelerates Hive using GPU, and its designs mainly consider performance and generality. We will integrate more techniques and extend our implementation to other big data systems in the future.

7 CONCLUSIONS AND FUTURE WORKS

In this work, we present GHive, an extension of Hive that utilizes GPU to accelerate OLAP query processing. GHive comes with three main technical components, i.e., data model gTable, GPU-based operator library Panda, and a cost-based job scheduler. gTable stores tables in column-based format and ensures that each column is consecutive in memory for efficient CPU-GPU data movement. Panda reduces the data movement overhead with an indexing-based processing model and achieves generality and extendibility with sensible implementations. The scheduler utilizes cost models motivated by profiling results to make judicious job placement decisions. Experiment results show that GHive outperforms vanilla Hive in both query processing time and operating cost, especially for computation intensive queries. As an initial attempt to utilize CPU-GPU heterogeneous computation, the outcomes of the GHive project are encouraging. However, we still pan to improve GHive in the following aspects.

Consider GPU execution in query optimization. Currently, we use the optimizers in Hive, which do not consider CPU-GPU heterogeneous computation. A GPU-aware optimizer may avoid grouping computation intensive operators and I/O intensive operators into the same job such that the resulting jobs are more suitable for GPU execution. The optimizer should also reduce the amount of data transfer between CPU executed jobs and GPU executed jobs.

Dynamic Scheduling. Dynamic scheduling, which makes the decision to place jobs on CPU or GPU at runtime, can be used to adapt to the availability of GPUs. Dynamic scheduling also helps to handle large errors in cardinality estimation—cardinality estimation can be made more accurate when some jobs are executed and the job placement decisions (made by the cost models) will be more sensible using the updated estimations at runtime.

More flexible execution patterns. GHive moves data back to CPU memory after executing a job on GPU even if its downstream jobs are also executed on GPU. Although this design simplifies data management, letting the downstream jobs continue on the loaded data is more efficient. In addition, GHive imposes a barrier for each GPU executed job as the job only starts when all input data are collected. To pipeline the jobs, a GPU executed job may receive and process input data in batches as the CPU executed jobs in Hive.

Data compression. The data movement overhead is a major obstacle for reaping the benefits of GPU execution. The column-based format of gTable enables simple and effective data compression schemes such as delta encoding and run-length encoding, and there are researches developing SQL operators that work on directly compressed data.

Extend to more data analytics systems. Besides Hive, we also run other data analytics systems such as Flink and Spark. The success of GHive motivates us to support CPU-GPU heterogeneous computation in these systems. The generality of Panda makes it easy to run the tasks of these systems on GPU but the interaction between CPU and GPU needs to be carefully designed for high efficiency.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and the shepherd Krishna Kantikiran Pasupuleti for their constructive comments and insightful suggestions on this paper. This work was supported by the Guangdong Provincial Key Laboratory (2020B121201001), Guangdong Basic and Applied Basic Research Foundation (2021A1515110067) and a research gift from Huawei. Dr. Bo Tang and Dr. Xiao Yan are the corresponding authors.

REFERENCES

[1] 2022. *Apache Arrow*. https://arrow.apache.org/
[2] 2022. *Apache Flink*. https://en.wikipedia.org/wiki/Apache_Flink
[3] 2022. *Apache Hadoop*. https://hadoop.apache.org/
[4] 2022. *Apache Spark*. https://en.wikipedia.org/wiki/Apache_Spark
[5] 2022. *BlazingSQL*. https://blazingsql.com
[6] 2022. *CUB*. https://nvlabs.github.io/cub/
[7] 2022. *GHive*. https://github.com/DBGroup-SUSTech/GHive
[8] 2022. *IBM Spark GPU*. https://github.com/IBMSparkGPU/GPUEnabler
[9] 2022. *libcuDF*. https://github.com/rapidsai/cudf
[10] 2022. *MPS*. https://docs.nvidia.com/deploy/mps
[11] 2022. *NVML*. https://developer.nvidia.com/nvidia-management-library-nvml
[12] 2022. *OmniSciDB*. https://www.omnisci.com/platform/omniscidb
[13] 2022. *rapids*. https://rapids.ai
[14] 2022. *Thrust*. https://github.com/NVIDIA/thrust
[15] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5 (2013), 197–280.
[16] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*. 221–230.
[17] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). 1891–1906.
[18] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. 2019. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *SIGMOD*. 1773–1786.
[19] Cen Chen, Kenli Li, Aijia Ouyang, and Keqin Li. 2018. Flinkcl: An opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. *IEEE Trans. Comput.* 67, 12 (2018), 1765–1779.
[20] Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. 2018. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data. *TPDS* 29, 6 (2018), 1275–1288.
[21] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB* 12, 5, 544–556.
[22] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in gpu-accelerated analytical engines. In *CIDR*.
[23] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *CIDR*.
[24] Jeffrey Dean and Sanjay Ghemawat. 2010. MapReduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.
[25] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2007. GPUQP: query co-processing using graphics processors. In *SIGMOD*. 1061–1063.
[26] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *SIGMOD*. 1603–1618.
[27] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *PVLDB* 13, 6 (2020), 884–897.
[28] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* 6, 9 (2013), 709–720.
[29] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *PVLDB* 6, 9 (2013), 709–720.
[30] Eugene Gorbatov Howard David, Rahul Khanna Ulf R. Hanebutte, and Christian Le. 2010. RAPL: memory power estimation and capping. In *International Symposium on Low Power Electronics and Design*. 189–194.
[31] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *SIGMOD*. 1235–1246.
[32] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*. 947–960.
[33] Tomas Karnagel, René Müller, and Guy M Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@VLDB* 8 (2015), 20.
[34] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. 2011. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*. IEEE, 25–36.
[35] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *PVLDB* 14, 12 (2021), 2999–3013.
[36] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.
[37] Zhila Nouri Lewis and Yi-Cheng Tu. 2022. G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing. *TKDE* 34, 3 (2022), 1243–1257.
[38] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *TPDS* 31, 1 (2019), 94–110.
[39] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *PVLDB* 9, 14 (2016), 1647–1658.
[40] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *SIGMOD*. 1633–1649.
[41] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. 283–294.
[42] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. 237–252.
[43] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on gpus. *PVLDB* 14, 2 (2020), 202–214.
[44] Johns Paul and Bingsheng et al. He. 2020. Revisiting hash join on graphics processors: A decade later. *Distributed and Parallel Databases* 38, 4 (2020), 771–793.
[45] Syed Mohammad Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR*.
[46] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*. 1357–1369.
[47] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for

database analytics. In *SIGMOD*. 1617–1632.

[48] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on gpus. In *ICDE*. 698–709.

[49] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009), 1626–1629.

[50] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*. 996–1005.

[51] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*. 1–16.

[52] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 44–54.

[53] Haicheng Wu, Gregory F. Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 44.

[54] Long Xiang, Bo Tang, and Chuan Yang. 2019. Accelerating exact inner product retrieval by cpu-gpu systems. In *SIGIR*. 1277–1280.

[55] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB* 6, 10 (2013), 817–828.

[56] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *IEEE Big Data*. 273–283.

[57] Yansong Zhang, Yu Zhang, Jiaheng Lu, Shan Wang, Zhuan Liu, and Ruichen Han. 2020. One size does not fit all: accelerating OLAP workloads with GPUs. *Distributed and Parallel Databases* (2020), 1–43.