The Art of Balance: A RateupDB^{TM} Experience of Building a CPU/GPU Hybrid Database Product

Rubao Lee
Rateup Inc.
Chengdu Sichuan, China
rubao.lee@rateup.com.cn

Minghong Zhou
Rateup Inc.
Chengdu Sichuan, China
minghong.zhou@rateup.com.cn

Chi Li
Rateup Inc.
Chengdu Sichuan, China
chi.li@rateup.com.cn

Shenggang Hu
Rateup Inc.
Chengdu Sichuan, China
shenggang.hu@rateup.com.cn

Jianping Teng
Rateup Inc.
Chengdu Sichuan, China
jianping.teng@rateup.com.cn

Dongyang Li
Rateup Inc.
Chengdu Sichuan, China
dongyang.li@rateup.com.cn

Xiaodong Zhang
The Ohio State University
Columbus OH, USA
zhang@cse.ohio-state.edu

ABSTRACT

GPU-accelerated database systems have been studied for more than 10 years, ranging from prototyping development to industry products serving in multiple domains of data applications. Existing GPU database research solutions are often focused on specific aspects in parallel algorithms and system implementations for specific features, while industry product development generally concentrates on delivering a whole system by considering its holistic performance and cost. Aiming to fill this gap between academic research and industry development, we present a comprehensive industry product study on a complete CPU/GPU HTAP system, called RateupDB. We firmly believe "the art of balance" addresses major issues in the development of RateupDB. Specifically, we consider balancing multiple factors in the software development cycle, such as the trade-off between OLAP and OLTP, the trade-off between system performance and development productivity, and balanced choices of algorithms in the product. We also present RateupDB's complete TPC-H test performance to demonstrate its significant advantages over other existing GPU DBMS products.

PVLDB Reference Format:
Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The Art of Balance: A RateupDB^{TM} Experience of Building a CPU/GPU Hybrid Database Product. PVLDB, 14(12): 2999-3013, 2021.
doi:10.14778/3476311.3476378

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476378

1 INTRODUCTION

In the past decade, GPU-accelerated database systems have grown up rapidly from an early stage of concept proofing (e.g. [22]), inaugurating and prototyping (e.g. GDB [48], GPUDB [115], Ocelot [53]) and many individual projects in the academic community (e.g, [18, 28, 44, 61, 78, 90, 95, 98, 99, 101, 105, 113]), to the stage of making products in database industry for large-scale deployments for various critical applications. Several commercial GPU DBMSs have been built, either by GPU-customized designs, e.g. Kinetica [6], OmniSci [9] (formerly known as MapD [84]), SQream [10], or by extending existing database systems, e.g. the PostgreSQL-based Brytlyt [2] and HeteroDB [4]. The design space for a GPU database is large, where many optimizations can be done to offer multiple possibilities to improve performance. However, the system improvement in many research projects is not necessarily achieved for the overall performance, but for isolated cases. Based on our development experience of RateupDB, a high performance GPU database product, we demonstrate that the essential issue is to achieve an overall system balance by trading-off multiple important performance factors. Only in this way, we are able to achieve high performance, scalability and sustainability for a GPU database.

1.1 The Rise of GPU DBMSs

The rise of GPU database systems is mainly driven by real-world application requirements. As the human society has entered the *data computing* era, which is driven by increasingly more and diverse applications supported by new technologies and innovative concepts in the society, such as mobile computing, digital content sharing, shared economy and others, new application requirements are often beyond the scope of functionalities provided and optimized by conventional database systems. Based on our collaborations with data industry practitioners and customers, we summarize the new requirements into the following three categories.

Requirement for Real-Time Business Insights: This is the most important application need for users who need to obtain instant business insights by applying various analytics on timely updated data (e.g., [33, 39, 106, 119]). A typical scenario is the real-time vehicle position management as represented by the cases of AresDB used in Uber [5] and Kinetica used in USPS [12]. We have learned from a major global service company owning several millions of operating vehicles that dynamic route planning and deviation rate analytics based on real-time vehicle positions are not only necessary to lower operational costs and improve user experiences, but also vital for drivers and passengers to benefit from on-time malicious behavior detection for criminal precaution.

Requirement for Extreme Computing Power: The explosive trend to process an increasingly huge amount of non-structural data seriously challenges the limited computing power in CPU-only database systems, particularly in this post-Moore's Law era. We have witnessed a number of daily production cases that demand ultra high-performance by hardware-accelerated solutions to process unconventional and huge data sets in relational databases, which are critically important for applications including but not limited to DNA sequencing [14, 80], spatial data analytics [16, 35, 110], and 3D structure analysis [79, 96]. We have surprisingly observed that users still prefer to use a database product to manage their data, although they are often unsatisfied with database system's performance for their tasks. This is because databases have an advantage of providing simple interfaces for users. If they use multiple fractional software tools, they may have to manually manage them.

Requirement for One-Stop Data Management: With the demand of real-time data analytics (from simple aggregations to complex statistics analytics), users often demand to use a one-stop solution to unify data writes, query processing, and machine learning. Compared to the commonly used two-stage solution (e.g., RocksDB+Spark [29, 118]), a single database (e.g. [5][12]) can significantly save users' cost in usage and management, and can avoid data transfers between separate systems. Other examples are Teradata Vantage [71] and Google BigQuery cloud service [1] that embed machine learning functionalities into their relational query execution engines; thus the need of using a separate machine learning system for the in-database data is removed. However, the requirements of performance isolation and service-level agreement bring new challenges for the implementation of one-stop databases.

Rapid Hardware Advancement: The rapid development of hardware technology allows GPUs to provide increasingly parallel computing power and large memory space. With the NVLink protocol [8], a single server can provide a fast-accessing memory pool of several hundreds of GBs to feed GPU cores for data processing. Therefore, such a hardware advancement trend is creating a *GPU-Style In-Memory Computing* environment, which has appeared in recent database research work (e.g., [116][99][90]).

More than ten years ago (in 2010), when RAMCloud was proposed to realize the concept of all-in-memory data storage [89], the two key factors used to make the case were 64GB DRAM Capacity/Server and \$60 Cost/GB DRAM. Today, such a concept has become the reality for data processing, e.g., in main-memory database system (e.g. [34, 64, 120]). Similarly, today's GPU hardware capabilities have already exceeded or stayed at the same level for the configurations in RAMCloud in both capacity and price/capacity. Even a single Nvidia A100 can have 80GB memory, which is bigger than the DRAM capacity per server in RAMCloud. Regarding the price, public GPU cards on Amazon.com (PNY NVIDIA Quadro RTX 8000 with 48GB GDDR6 memory for \$4,997 and EVGA GeForce RTX 3090 WITH 24GB GDDR6 memory for \$2,499) have already provided a memory price at around \$100 Cost/GB – **even if we only count the final price to the memory part**. The commodity GPU development gives us a promising landscape, and we can optimistically state that GPU's computing power will be best utilized with fast data accessing in large memory space.

1.2 Targeted Issues in Building a GPU DBMS

Although many academic research efforts have been made for optimizing GPU database processing, (e.g, [18, 28, 44, 61, 90, 95, 98, 99, 101, 105]),they often focused on a single aspect, without giving a complete reference for building a real database product. Here are the three issues we attempt to address in this paper.

The Issue of Algorithm Choices: Various algorithms (e.g., hash join vs sort merge join) are available, but how to make a right selection among them is a key issue for an industry product.

The Issue of Simple Data Analytics: In existing GPU database literature, we often find that performance evaluations are conducted by SSB-like (Star Schema Benchmark) workloads. However, an industry product demands intensive performance evaluations by complex data analytics workloads, such as TPC-H queries that requre much more practical SQL features [11].

The Issue of Isolated Environments: In existing GPU database literature, we often find that the query performance is evaluated in an isolated environment, assuming there is no companion data modifications. However, daily database operations in practice are unavoidably involved with both reads and writes.

1.3 Contributions

This paper aims to address the above mentioned issues by presenting the basic structure, design choices and performance insights into RateupDB. Attempting to fill the gap between academic research and product development, we make the following contributions.

Making Right Choices by Taking Balanced Considerations: We introduce a set of macro-level design choices for implementing RateupDB, and why we make these choices. We present implementation techniques for several important parts, especially on how to combine query executions and transaction processing.

Performance Insights into RateupDB by Complex Data Analytics in a Production Environment: We present RateupDB's performance for a complete TPC-H test: (1) by using the official benchmark specification and (2) by strictly satisfying the test requirements. Our comprehensive evaluation results confirm the value of the system balancing principle in RateupDB.

2 THE ARCHITECTURE

We first discuss several identified macro-level balancing points, which lay the foundation for RateupDB's overall design strategy.

2.1 Key Balancing Points in System Building

RateupDB is a Heterogeneous Hybrid Transactional and Analytical Processing (simply called Heterogeneous HTAP, or H^2TAP [18][95], and we simplify the term as HTAP in this paper) database system, which is designed to best utilize CPU, GPU, and large DRAM memory for co-running hybrid workloads. To achieve the goal for an industry product, the development of RateupDB is focused on the following three key balancing points, which are related to both technical challenges and engineering costs.

Balancing Point 1: Transaction vs Query. A major goal for a HTAP system is to optimize both transaction processing and query execution in a shared database platform while providing performance isolation for satisfactory user experiences in quality of service (QoS). This point is largely related to major system strategies including hardware resource partitioning, data store formats and access methods, and concurrency control methods. Making balanced performance trade-offs in a large design space is challenging due to various conflicting factors (e.g., the RUM conjecture [19][93] or multi-source integration analytics [94]).

Balancing Point 2: Performance vs Engineering Cost. Unlike an academic research project, the development of an industry product must consider engineering cost that is a constraint from both the budget of financial resources and the requirement of production deadlines. Therefore system design choices and various algorithm adoptions must balance multiple factors including performance, adaptiveness, robustness, and implementation cost (e.g., the independence degree to other system components). A typical example for this balancing point is the choice of GPU join algorithms, which will be discussed later in this paper.

Balancing Point 3: Performance vs Portability. To achieve the goal of both high-performance and being-portable in software development is challenging. Specifically, in high performance computing, e.g., [92][46], we often aim at achieving two goals that are conflicting: (1) to best utilize all the features in both hardware platforms and software libraries, and (2) to maximize software adaptiveness if certain hardware features are not universally available. This balancing point is important for the development of RateupDB that often requires GPU hardware-conscious algorithms (e.g., the memory management problem to be discussed later in this paper).

2.2 An Architectural Overview

Figure 1 shows an architectural overview of RateupDB. We summarize RateupDB's main features as follows.

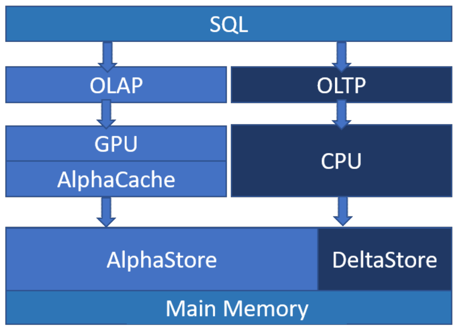

Figure 1: An Overview of RateupDB Architecture.

Hardware-guaranteed Performance Isolation: A major challenge of building HTAP systems is to minimize performance interference between co-running transaction workloads and analytical workloads because the two types of workloads have distinct execution patterns, SLA (Service-Level Agreement) requirements, and optimization policies. Achieving the dual goal of performance isolation and frequent data-update is challenging in CPU-only HTAP systems [95]. However, a GPU-based HTAP system has its unique advantages to address this issue due to its capability of assigning and running different workloads in separate hardware devices. As shown in Figure 1, RateupDB uses CPU to execute OLTP workloads (i.e., insert/delete/update) and uses GPU to execute various queries of OLAP workloads in the shared data stores. Because the CPU and the GPU are separate hardware devices, performance isolation can be guaranteed if corresponding software is well defined.

MVCC-based Dual Data Stores: Data store as a persistent repository plays an important and foundation role to both query execution and transaction processing, particularly when the two workloads are co-running together. Since RateupDB is built on both CPU and GPU, the design and implementation of data store must be in a way so that (1) query execution on GPU is efficient and (2) performance interference is minimized. To achieve the two goals, RateupDB uses "dual stores" in main memory, which consists of two parts: AlphaStore and DeltaStore (see Figure 1). For a given time of the database, AlphaStore stores the existing data in the database for query processing, while DeltaStore stores the changes made to the database by new transactions. A Multiversion Concurrency Control (MVCC) is utilized to manage the two stores cooperatively. In this way, the GPU device can process OLAP queries independently by obtaining a snapshot of the database from combining AlphaStore and DeltaStore, while the CPU can execute transactions to modify the database states in DeltaStore. In addition, RateupDB uses a part of GPU device memory (called AlphaCache) to cache frequently used data items in AlphaStore in order to avoid unnecessary data transfers from host memory to GPU device memory. Currently, both AlphaStore and DeltaStore are in column store formats by which each column of a table is stored independently.

3 DATA FORMAT

3.1 Data Store Format Matters!

Although the basic data format for a relational table is either row store or column store, there are multiple choices in the design space, e.g. (1) choosing one of the basic stores, (2) mixing them to form a hybrid format (e.g. [15, 51, 58]), and (3) combining the basic stores using multiple data stores. Table 1 lists several typical database systems for each category of data stores, which shows diverse design choices in different database systems.

3.2 The Necessity of Dual Stores

As a HTAP database system, RateupDB takes a dual store approach (column+column) for several reasons. A conventional wisdom is that row store is more suitable for transaction workloads (e.g., PostgreSQL) while column store is more suitable for analytical workloads (e.g., MonetDB [27]). But for HTAP systems that have to handle both transaction and analytical workloads, neither row store nor column store is sufficient. Thus several HTAP systems (Hyper [62], SAP HANA [100], Caldera [18], BatchDB [81]) use different ways of combining the two stores. Basically, the four systems fall into two categories: (1) Single-Store: allowing users to

Table 1: Examples of data stores in several DBMSs.

Format/DB	Postgres	MonetDB	HyPer	SAP HANA	Vertica	Caldera	BatchDB	RateupDB
Unified Store: Row	Yes		Choice			Choice		
Unified Store: Column		Yes	Choice			Choice		
Unified Store: Hybrid						Preferred		
Dual Store: Row+Row							Yes	
Dual Store: Row+Column				Yes (L1-L2)	Yes		Future	Tried
Dual Store: Column+Column				Yes (L2-Main)	Tried			Yes

choose either row store, column store, or a hybrid store; (2) Multi-Store: combining two or even three stores to best serve transactions and queries. CPU-based HyPer and GPU-based Caldera are typical examples in the first category, while SAP HANA and BatchDB are in the second one. For the first category, compared to HyPer, Caldera's performance results suggested that a hybrid data store be a more effective choice than pure row store or pure column store. For the second category, SAP HANA uses a three-level stores (L1 Delta, L2 Delta, and main store), where the first level is in row store for absorbing transactions and the last two levels are in column store for query processing. BatchDB (as well as TiDB [59]) uses separate stores (by replicas) to isolate OLTP and OLAP. Oracle Database In-Memory (ODIM) [70, 91] is also in this category, which takes a dual format in-memory stores (column store for OLAP and row store for OLTP – caching disk data).

The unified single-store approach and the separate dual-store approach in Table 1 have both pros and cons. First of all, to maintain separate stores needs extra cost to manage and merge multiple data stores. However, separating stores for transactions and queries can minimize the interference on the shared data, which is particularly suitable for CPU-GPU heterogeneous database systems, where queries are executed on GPU and transactions are executed on CPU. Ideally, when GPU begins to execute read-only queries, it should minimize the interference to CPU by avoiding making CPU execute too much on format parsing or on data preparations. Furthermore, when CPU executes transactions, its write operations and synchronization-related operations (locks or latches) should not interfere with GPU processing. A dual store, if managed well, is an effective solution to achieve these goals.

3.3 Why Column+Column?

Now the question is what data formats should be used in a dual store. First, for analytical tasks, column store is the best choice for its various performance advantages [103][13]. Therefore, almost all the above mentioned HTAP systems use column store for their analytical part. The only exception is BatchDB [81], which uses a combination of row+row store because the purpose is to demonstrate the necessity of use separate stores for OLTP and OLAP. Nonetheless, the team of BatchDB admits the advantage of column store for analytics, and plans to implement it in future versions.

Second, for the transaction tasks, we have observed that not all systems consistently use row store despite its claimed advantages for OLTP workloads. SAP HANA applies a hybrid approach to bridging the transaction tasks and the analytical tasks by applying a row-store L1-delta and a column-store L2-delta in front of its column-oriented main memory store [100]. Two other system cases (Hyper and Vertica) demonstrate the possibility of using column stores for absorbing transaction writes to the database. **HyPer [62]:** "*HyPer can be configured as a row store or as a column store. For OLTP we did not experience a significant performance difference.*"**Vertica [72]:** "*The WOS has changed over time from row orientation to column orientation and back again. We did not find any significant performance differences between these approaches and the changes were driven primarily by software engineering considerations.* "

The fundamental reason of using column store for transactions is that it can greatly accelerate query processing and table merging operations, while its performance reduction for OLTP is negligible or acceptable. Like Vertica, RateupDB began with a row+column dual store, but it had been quickly changed to a column+column dual store in order to accelerate analytics workloads while keeping transactions at a level of acceptable performance.

3.4 RateupDB's Data Store

AlphaStore: AlphaStore (meaning the main store) is a column-oriented data store loaded into main memory from disks, by which each table column is stored in continuous data chunks. When stored on disks, AlphaStore does not contain any MVCC information. After loading, all tuples are read-only and visible to all later transactions, unless they are deleted, which would be recorded by DeltaStore. Essentially, when a query is executed, the data in AlphaStore will be transferred into GPU for query processing. RateupDB uses a zone in the GPU device memory, called AlphaCache (see Figure 1), to store frequently used data in AlphaStore by an LRU algorithm.

DeltaStore: DeltaStore is a MVCC-based column store, which records the changes to AlphaStore after each transaction is finished. For an insert command, the newly inserted data will be appended into corresponding columns of DeltaStore. For a delete command, the IDs of rows being deleted will be recorded by DeltaStore into a delete vector. For an update command, it is essentially converted into an insert and a delete operation. Detailed concurrency control of the stores will be presented in Section 5.

4 THE GPU QUERY PROCESSING

RateupDB uses a GPU-based query execution engine. In this section, we focus on the following three aspects. (1) **Engine Structure:** It essentially determines how a series of relational operations in a query planning tree, are connected and executed on GPU devices; (2) **Algorithm Choices:** It concerns how to best utilize GPU hardware to implement each concrete algorithm for various operators; (3) **Complex Query Handling:** It addresses issues on how to execute complex SQL queries with various subquery forms.

4.1 On Query Engine Structure

RateupDB's query engine structure applies an approach of *operator-at-a-time*, which is inherited from its early prototype GPUDB [115]. Basically, all the operators in a query plan tree are finished in a sequential order according to the tree structure. For each operator, after all its input data are transferred into GPU memory, a set of GPU kernels are executed, while the final output results of the operator are kept in the GPU device memory for consequent operator executions. Although such a model has its advantage of universally executing all SQL queries, it may not effectively execute multiple operators that have correlations and thus can be finished in a combination way. To address the problem, RateupDB takes an approach of *combinational operator* to optimize certain important OLAP query scenarios. RateupDB uses *Star Join* and *Self Operator* to achieve the goal of best utilizing GPU computing resources while minimizing data transfer overheads, which will be discussed later.

4.2 On Join and Grouping

A significant effort for RateupDB is focused on how to effectively execute various relational operations in GPU, particularly for join and grouping/aggregation. There has been a long history of debating on (1) how to implement the two operations and (2) what to choose: sort-based algorithms vs hash-based algorithms. A thread of existing research work to address the above two questions on multi-core or many-core CPUs often gives different conclusions. However, a widely accepted conclusion is that a complex cost model by considering both data distributions and hardware parameters is needed to compare these two categories of join algorithms. We face a challenge on how to build a query engine by taking prior research results into consideration in a large algorithm design space.

4.2.1 Sort vs. Hash Revisited. We first present a brief survey of existing results for the issue of sort vs hash in both CPU and GPU databases, which lays a foundation for our approach in algorithm design and implementation. Kim et. al [63] proposed two SIMD-optimized parallel join algorithms: a radix hash join algorithm and a sort-merge join algorithm that uses bitonic merge networks and multiway merging to implement the underlying sort operation [30]. Their performance analysis results project that the sort-merge join could outperform hash join for future many-core hardware with wider SIMD and smaller per-core memory bandwidth. Blanas et. al [25] showed that non-partitioning hash join (using a shared hash table) can outperform more complex partitioning-based hash join algorithms including radix hash join. Albutiu et. al [17] proposed NUMA-optimized, massively parallel sort-merge join algorithms that use range-partitioning-based merge-join techniques with a combination of multiple sorting algorithms including radix sort and quick sort. Their performance comparisons show that sort-merge join is faster than both radix hash join and non-partitioning hash join. Balkesen et. al [21] proposed both optimized radix hash join (by reducing cache misses and TLB misses) and SIMD-optimized sort-merge join whose sorting utilizes sorting networks and bitonic merge networks. In addition to demonstrating that their join implementations are faster than the ones in both [25] and [17], they also showed that their hash join is faster than their sort-merge join. However, for large workloads, the performance is comparable.

Shifting from multicore to GPU brings a new debate on sort vs. hash. He et. al [49] compared radix hash join and sort-merge join using bitonic sort and quick sort on early GPU hardware, and concluded that radix hash join is faster than sort-merge join. Yuan et. al [115] proposed a non-partitioning hash join on GPU for star-schema queries. A recent GPU related study by Shanbhag et. al [99] also used such a non-partitioning strategy and indicated that an advantage of non-partitioning hash join compared to radix hash join is its effective usage in pipelined multiple joins. Despite radix hash's single-join performance advantage over non-partitioning hash join [21], its performance advantage compared to GPU-optimized sort-merge join is weakened, considering GPU Merge Path algorithm [45][87] for fast merging, and other radix sorting algorithms [102]. Rui and Tu [98] implemented an improved radix hash join by avoiding twice hash probing and a sort-merge join that uses GPU Merge Path for both the sorting stage and the merge-join stage. Their performance analysis showed that sort-merge join outperforms radix hash join on GPU. It is worth mentioning that their general-purpose merge sorting is not the fastest radix-based sorting [102] that has limitations for arbitrary data types. Siloulas et. al [101] implemented hardware-conscious radix hash join and showed its speedup over the non-partitioning hash join.

4.2.2 The RateupDB Approach. Ideally, all above mentioned join algorithms should be implemented in a DBMS, and this system further utilizes a query optimizer with precise cost models to select the right algorithms for a given query. However, it is a non-trivial task to implement such a query optimizer (e.g., [74]) that involves a set of execution components including data distribution estimation and runtime statistical information collecting. Nonetheless, its value in practice may often be insignificant [77].

To balance the database performance and engineering efforts, our RateupDB approach to implementing join algorithms is based on the following three principles. (1) **Implementing a default, general-purpose join algorithm.** We use sort-merge join to serve this purpose. Its sort stage uses both radix sorting and merge path sorting, depending on the join key characteristics, and its merge-join stage uses the merge path algorithm. (2) **Optimizing star joins dominated data warehouse queries.** We use non-partitioning hash join for this purpose. Furthermore, it provides a star-hash join for multiple joins based on the binary non-partitioning hash join. (3) **Using simple rules to select join algorithms.** We do not use any cost models, but use simple rules based schema information only. Our join algorithm selection process is illustrated by Figure 2.

The reasons why we take the sort-merge approach for the default join algorithm instead of radix hash join are both performance related and development-cost related. **Performance Considerations:** As pointed out in [98], if both are optimized by utilizing GPUs, sort-merge join is faster than radix hash join. This conclusion is consistent with the results provided by [21], which shows sort-merge join is more favorable for large input tables. **Engineering and Development-Cost Considerations:** Sort-merge join has certain advantages over radix hash join for both robustness to handling data size variations and its symmetry nature of removing the necessity of a query optimizer to designate which table is for hash building or probing. Furthermore, the sort stage, which is a general parallel computing problem, can be separated from

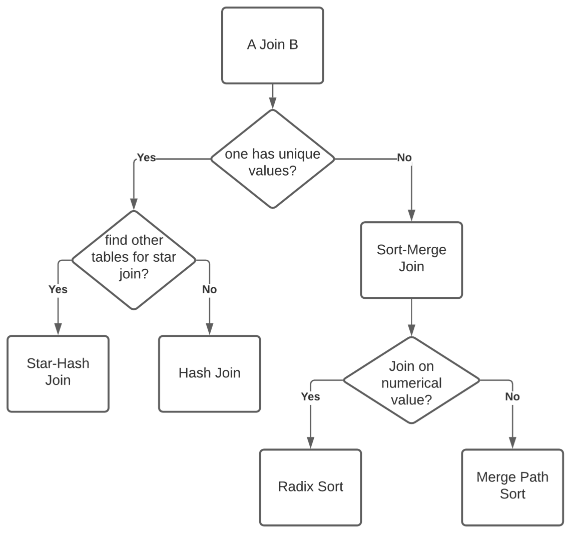

Figure 2: A Selection Process of Join Algorithms.

the merge join stage so that it can be independently optimized by CUDA engineers without any database background knowledge.

For primary-foreign key join, we use non-partitioning hash join [115][99]. Compared to radix hash join, the basic advantage of non-partitioning join is that it allows a pipelined star join execution among one fact table and multiple dimension tables. In our implementation, each tuple from the fact table probes the hash tables of all the dimension tables. Our performance measurements show that such a technique is critically important for improving performance of executing typical warehousing queries, such as TPC-H queries. Even for the binary join, where the fact table is often much larger than the dimension table, the overhead of radix hash join's partitioning step for the fact table is unnecessarily high considering its unavoidable random memory accesses [21][98].

For sort-merge join, we take a hybrid approach for the sort stage. Its default choice is to use radix sort, if the data values are suitable for being sorted lexicographically, such as a integer sorting. Essentially, we use the CUDA CUB header library [3] as the working engine for radix sort. Because CUB can only execute sorting on C++ numeric primitive types, it cannot be used to sort complex data types, such as Decimal that often has a struct type wrapping internal data representations. For such data types, we use a general merge sort algorithm based on GPU Merge Path [98].

4.2.3 On Grouping. Like join, processing grouping in a GPU database can also be hash-based (e.g., GPUDB [115]) or sort-based (e.g., CoGaDB [28]). By default, RateupDB uses a sort-based grouping strategy. First, sort-based grouping outperforms hash-based grouping when the number of groups is large. Prior research work [61][105] reported that for grouping, when there are less than 200,000 groups, the hashing algorithm is faster. However, once the hash table is too large for L2 cache to store, its performance is significantly lower than that of the sorting algorithm. Such a phenomenon is actually similar to the multicore CPU join scenario [21]. Second, sort grouping can be implemented easily and better optimized than hash grouping, considering that (1) sorting can be implemented separately, and (2) it is a non-trivial task to implement an efficient hash table on general data sets to minimize or remove collision completely [53][105].

4.3 On Subquery Processing

Subquery processing is important for SQL performance in database products, such as in SQL Server [37] and in Oracle Database [23]. RateupDB does not use general-purpose subquery processing algorithms (e.g. [86] [41]). However, we implement commonly used subquery unnesting techniques that can transform subqueries into various joins and aggregations (e.g., the Kim method [65]), which allows RateupDB to efficiently execute TPC-H-style queries.

4.3.1 An Example. Although giving a comprehensive overview of subquery processing is out of the scope of this paper, we introduce how we optimize an important subquery type: EXISTS and NOT EXISTS. The following code is a part of TPC-H Q21, which essentially uses two subqueries to filter out some records from *lineitem*. We take a typical subquery unnesting technique to execute the query by: (1) using semi-join to execute the EXISTS subquery and (2) using anti-join to execute the NOT EXISTS subquery.

```
SELECT  ... FROM lineitem l1, ...
WHERE l1.l_receiptdate > l1.l_commitdate
      AND EXISTS ( SELECT * FROM lineitem l2
          WHERE l2.l_orderkey = l1.l_orderkey
                AND l2.l_suppkey <> l1.l_suppkey)
      AND NOT EXISTS (  SELECT * FROM lineitem l3
          WHERE l3.l_orderkey = l1.l_orderkey
                AND l3.l_suppkey <> l1.l_suppkey
               AND l3.l_receiptdate > l3.l_commitdate)
```

4.3.2 Semi/Anti-join and Self. Implementing semi- and anti-join is similar to equi-join, using either hash-based or sort-based algorithms. Because each of the left table's tuple is outputted either once or not, the output buffer can be pre-allocated with the length of the left table. Parallel threads can use AtomicAdd to write the results to correct positions, without the need of twice probing in hash equi-join or a prefix scan in sort equi-join.

However, considering the query semantics in Q21, naively executing the filter and two joins is slow because all the operations are only conducted on the same table. To exploit such a correlation [75][57], we implement a Self operator, which combines multiple operations into a common stage. This is similar to the Super-Operator concept in Microsoft SCOPE [76] and HorseQC [43]. If multiple operations on the same table have the same partitioning opportunity, i.e., the whole task can be partitioned into independent sub-tasks, then the Self operator will be enabled. In the same example, Self will partition *lineitem* by the shared join key *l_orderkey* and execute the three operations for each partition.

The Self operator can be hash-based or sort-based, depending on whether the partitioning column has only unique values. For the same example, Self is sort-based. Like sort-based grouping, the table is grouped first. Inside a group, Self will execute the required operations (by calling corresponding functions in other operator modules), similar to the GROUP concept used by Pig Latin [88] to allow users to execute arbitrary codes instead of only aggregations.

5 TRANSACTION PROCESSING

5.1 The Basic MVCC Method

We use a PostgreSQL-style Multiple-Version Concurrency Control (MVCC) to implement snapshot isolation in RateupDB. As illustrated in Figure 3, RateupDB uses a dual store mode, namely, AlphaStore and DeltaStore. Both stores are column store. AlphaStore is read-only and backed up by permanent storage devices, while DeltaStore only resides in main memory for both reads and writes.

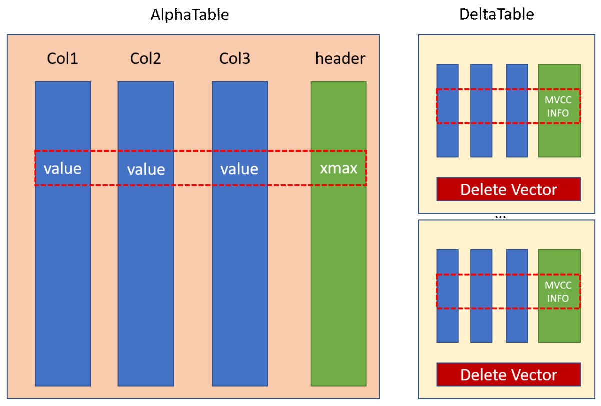

Figure 3: AlphaStore and DeltaStore.

When RateupDB is turned on, it loads AlphaStore from permanent storage devices into the main memory. For each table, AlphaStore contains only its columnar data without any MVCC information. After loading, the columnar data become read only. RateupDB will add a special *header* column for each table, which will be used to record MVCC information of each tuple. However, since AlphaStore is visible to all later transactions, the MVCC information stored in the header column is incomplete, which only records the *xmax* information (i.e., the transaction ID that deletes the tuple) without the *xmin* information (i.e., the transaction ID that creates the tuple).

DeltaStore stores data modifications by transactions, which can have multiple blocks. When one block is full, another block will be allocated and used, as shown in Figure 3. Although the overall design of DeltaStore is similar to PostgreSQL, it has two unique differences. First, it does not use a *page* concept because DeltaStore is only a main memory store [34]. Second, it uses a delete vector to record AlphaStore-related deletion information. When inserting a new tuple, its data will be appended into corresponding columns. When deleting a tuple, the tuple will be set *xmax* (same as from AlphaStore and DeltaStore), while its tuple ID (only for AlphaStore) will be appended into the delete vector. When updating an old tuple with a new tuple, a delete and an insert are executed. In DeltaStore, each new tuple and the delete record have complete MVCC information (e.g., *xmin* and *max* and other flags).

When executing a query, the CPU will execute a MVCC scanning of DeltaTable to generate two kinds of snapshot contents: (1) the IDs of tuples in AlphaStore that are invisible to the query (i.e., already deleted) and (2) the column data of all the tuples in DeltaStore that are visible to the query. When GPU needs to process a column, the CPU will send the two snapshot contents along with the original column in AlphaStore to GPU. In this case, the GPU does not need to handle any MVCC information, but only executes a simple kernel to filter out unnecessary tuples (like a filter kernel) from the original column and then combines the new data for consequent query processing. If the original column is already cached in the GPU device memory (AlphaCache), then only the two snapshot contents are transferred from the host memory to GPU. Considering the relatively much smaller size of DeltaStore than that of AlphaStore, such a design can efficiently accelerate query execution on the GPU side. However, MVCC scanning on the CPU side is not free of cost since the PostgreSQL-style MVCC implementation uses a unified store, where uncommitted tuples and committed tuples are mixed, and a complex visibility check for each tuple according to MVCC information (e.g., *xmin* and *xmax*) is unavoidable. Accelerating the MVCC visibility check is performance-critical, for example Netezza [42] uses FPGA to do the check with the purpose of avoiding transferring invisible tuples. In RateupDB, we implement a special MVCC invariant, aiming to accelerate the check.

5.2 Improvement for Fast Query Execution

As an implementation approach to snapshot isolation, how to implement MVCC's visibility check for a tuple can be different. In the original paper introducing snapshot isolation [24], requirements for physical implementations are not mentioned by the following statement: *"At any time, each data item might have multiple versions, created by active and committed transactions."*. Tuples written by uncommitted transactions should be invisible to other transactions. However, the invisibility check for such a tuple can mean two methods in different implementations: (1) a transaction can access the tuple and then finds it invisible; and (2) the transaction cannot access the tuple at all. The former means that all tuples are written into a public space no matter whether or not the transaction is committed (e.g., PostgreSQL), while the latter means that an uncommitted transaction can only keep its written tuples in its private space, and until it commits those tuples are installed into the public space. The second situation is implied in the re-defined snapshot isolation in [40], which stated that a transaction *"holds the results of its own writes in local memory store"*. Also, the second method is more similar to the Optimistic Concurrency Control (OCC) method [69] and other concurrency control algorithms and implementations (e.g., [108, 117]).

The first method was used in an early version of RateupDB, which has been changed to the second one to accelerate query executions. For transaction executions, the second method's disadvantage compared to the first one is that it has a *double write* problem, which causes an inserted tuple to be written twice in memory. However, its advantage is that it avoids scanning written tuples of uncommitted transactions, particularly from transactions that need to write a number of tuples in a batched mode, which is more common in an analytics-oriented environment. Furthermore, because DeltaStore only records tuples from committed transactions, RateupDB can organize DeltaStore by the order of committed transactions, by which each transaction commitment will be appended to DeltaStore. Therefore, when a query execution begins, it only needs to scan the tuples by and before the latest committed transaction and ignore any later tuples because they are invisible to the query. More importantly, the scan does not need to compare *xmin* or *xmax*. Of course, a tuple can still be invisible if it appears in the delete vector, where only records committed deletes stay. Another advantage of the second method is the easier merging of DeltaStore into AlphaStore. For a DeltaStore chunk that is already full, if the ID of the latest transaction that modifies the chunk has already been surpassed by current transaction ID, then it means that the tuples in the chunk is already visible to any future transactions, and therefore the chunk can be appended into AlphaStore.

6 CPU TASKS VS GPU TASKS

6.1 On Query Executions on CPU

Currently RateupDB does not support CPU OLAP queries, but relies on the CUDA-based GPU query engine only. Maintaining an additional CPU query engine is a non-trivial engineering task due to execution pattern differences between CPU and GPU. Our early GPUDB prototype supports both CUDA and OpenCL so that queries can be run on both CPU and GPU. However, we have learned from our development experience that using OpenCL in a commercial product may not be an effective choice based on a reliability consideration. Without using OpenCL, a CPU query engine totally different from a GPU CUDA query engine would require a large effort of software development.

RateupDB does allow CPU and GPU to work together to process some queries. This happens under two conditions: 1. When the GPU device memory cannot hold all the input data, a partitioning-based strategy is enabled. The CPU needs to execute necessary pre-partitioning, merging, and materialization for the data in main memory. 2. For certain SQL functions (e.g., the GROUP_CONCAT function in MySQL), if the output sizes of parallel threads are not predictable or they have non-uniform distributions, the function will be executed on the CPU because it is difficult to effectively pre-allocate GPU memory for each thread's output data.

Currently RateupDB uses GPU to execute the where condition part (which could have a subquery embedded) in an update statement. Such a strategy is suitable for a TPC-H-style workload that have two characteristics: (1) concurrent inserts, and (2) concurrent batch updates that select multiple rows from a pre-defined table of lookup keys. This approach benefits from avoiding index building on the CPU side, subject to an acceptable query latency for the updates. However, we are aware that a CPU-based index-scan is unavoidable for heavy OLTP workloads with a lot of point queries (i.e., key lookup). We plan to add such a feature in next version.

6.2 On Indexing

Currently RateupDB does not use any index for OLAP queries. But it uses a special indexing method to accelerate constraint check for OLTP workloads when inserting a new tuple into a table with a primary key or a foreign key. In the case of a large table, the insert operation will have to be dominated by the primary key check if no auxiliary indexing structure is available. Unlike a conventional database, for example PostgreSQL, which often utilizes a unified *B+tree* to serve the check, RateupDB uses a hybrid indexing method for the dual store. First, for DeltaStore, a hash index is built to quickly locate the tuple IDs according to the primary key. Second, for AlphaStore, due to its read-only nature, the column of the primary key is first sorted after being loaded, then whenever for a later primary check, a binary search on the sorted column is executed for ID locating. The sorting is actually executed on GPUs. Without building a hash table for AlphaStore, memory space is greatly saved while offering acceptable performance.

6.3 Further Possibilities

Currently the query optimizer and the transaction engine are on CPU only. This could be done in GPU but we did not do it for the following reasons. First, for the query optimizer, research efforts have been done to investigate possibilities of using GPUs to execute some critical query optimization tasks, such as selectivity estimation [52] and join order optimization [83]. Furthermore, recently machine learning based query optimization work (e.g., [66–68, 82, 104, 107, 112, 114]) have shed light on automatic query optimizing software using deep learning or reinforcement learning algorithms, which can greatly benefit from GPU's massively parallel computing power. However, it is still unclear how to best utilize those algorithms in a HTAP database with a balancing consideration among overall benefits, performance isolation, and engineering cost. Second, for the transaction engine that has distinct execution patterns from read-only OLAP query processing, prior studies [50] [121] [56] also demonstrated the possibilities of using GPUs to execute transactions or to build an underlying key-value stores. However, those systems have certain limitations (e.g., only supporting pre-defined stored procedures or only being a data store without a transaction support), thus they are not general-purpose transaction engines on GPU.

7 THE HETEROGENEOUS MEMORY

7.1 A Tale of Two Memories

Since the rise of general-purpose GPU computing, significant efforts in computer system research have been focused on solving problems caused by a tale of two memories: the host memory (DDR DRAM) for CPU cores and the GPU device memory (GDDR SDRAM) for GPU cores, which are physically separated, but connected by a communication link (PCIe or NVLink). Since the GPU cores can only directly access the GPU's device memory, its main disadvantages are summarized as follows.

Too small to hold a large data set: The problem of limited size of physical GPU memory is one of major GPU programming challenges. From the hardware and systems' perspective, various efforts have been made to add virtual memory supports to provide an illusion of unlimited memory space [73][109][122][20], or to use hardware compression algorithms to save memory space [32]. From the applications' perspective, the problem causes significantly additional programming efforts by partitioning large input data into multiple chunks for processing them separately, either by utilizing multiple GPU devices for parallel processing or by transferring data chunks between GPU memory and the host memory.

Too far to reach quickly: With two physically separated memories, the main bottleneck for GPU databases is the PCIe transferring, i.e., the so-called *"Yin and Yang"* problem [115]. A variety of optimization techniques have been proposed, such as (1) utilizing data compression to reduce transferred data size [38], (2) caching transferred data in GPU memory for data reusing [111],

(3) maximizing the data usage once transferred [43], (4) avoiding unnecessary data movement by lazy and/or shared transfers [95], and (5) overlapping transfers and computations to hide latency [97, 111].

Too hard to program: The combination of the two above problems makes early stage GPU programming hard due to the lack of a unified virtual memory that can release programmers from manually managing two separate memories. Since Nvidia Pascal GPU architecture (CUDA 8 for software) [47], a unified memory programming model using the ManagedMemory APIs becomes available, thus GPU programs can be developed with less memory constraints. However, significant performance degradation is still unavoidable in the case of GPU memory over-subscription due to uncontrollable page faults and thrashing [47][32].

7.2 The Design Choice

Considering the availability of multiple memory management methods for GPUs, a GPU database should balance multiple factors including single query performance, concurrent query throughput, and the cost of programming efforts. Before we introduce RateupDB's memory management implementation, we give the following two goals as the foundation of our design.

Maximized Single Query Performance: The development of RateupDB started with the GPUDB [115] prototype that treats GPU as a co-processor with explicit GPU memory allocation and data transfers. For each operator, a careful data partitioning step is needed to determine how much data can be processed at one time according to a predicted space allocation based on the running tasks and the GPU memory size. However the difficulty of precisely estimating the needed memory space makes this method hard to implement concurrent query processing.

Maximized Programming Simplicity: AlphaStore is fully implemented by the unified memory that is initially allocated using *cudaMallocManaged*. Thus, consequent query executions are directly implied on AlphaStore without any explicit data transfers. All the temporary data structures (e.g., intermediate results or hash tables) are also implemented in this way. Such a programming simplicity allows us to develop RateupDB without any measures on the available physical GPU size. However, uncontrolled data thrashing for large workloads can cause performance degradation.

7.3 Memory Management in RateupDB

With the consideration of the issues discussed earlier and possible design options, we take a hybrid approach for RateupDB's memory management. First, as a database server application, RateupDB partitions the GPU device memory into two zones: (1) the AlphaCache zone that is used to cache hot data in AlphaStore (currently a LRU algorithm is used for cache replacement) and (2) the work zone that provides space for kernel executions (i.e., storing necessary data structures and results of kernel executions). Second, the work zone is used in a way that needs explicit memory allocations and transfers (if needed) without using ManagedMemory APIs. Finally, RateupDB uses ManagedMemory only in some special situations where required memory space is not easily estimated, such as a *bad area* for hash table building even after several rehashing due to collision, or the output for a Cartesian product of two tables. Such cases are rare in practice, so that using ManagedMemory for them does not hurt performance, which can significantly reduce the programming effort.

RateupDB's hybrid memory management implementation is a result of balancing performance and implementation simplicity. First, as a major performance factor for server applications, locality in memory hierarchy plays the most important role for concurrent data accesses. However, letting CUDA's unified memory mechanism manage locality is unrealistic due to its inefficiency of handling typical memory system issues, such as thrashing [31]. Second, the approach to maximizing single query performance completely ignores inter-query locality so that each query can use all the GPU device memory for its peak performance. However, this approach is not optimized for the overall throughput considering the data reusing opportunities. Third, since the work zone is exclusively used during a query piece's execution, explicit memory management without involving any automatic mechanism by ManagedMemory can highly guarantee query performance. Finally, totally disabling the usage of the unified memory is unnecessary and inefficient. For example, pre-allocating space for data with unknown sizes in many algorithms is often achieved by a *twice execution* approach, which means that the algorithm (e.g., a hash probing operation) executes the first time to determine how much space could be, and then with the allocated memory size in the second time execution to write the output results [49, 98]. ManagedMemory can simplify the approach that dynamically allocates memory.

7.4 On the size of AlphaCache

RateupDB uses a configurable parameter to designate the size of AlphaCache. As shown in existing studies on auto-tuning database system parameters [36][55], reaching to a sweet spot for optimization of both the cache size and performance is hard and is often empirically based. For RateupDB, the size of AlphaCache is related to multiple factors, and the dominating ones include the GPU memory size, the working set size, and the sizes of the most important data structures (e.g., commonly used hash tables for either join or aggregation). By default, RateupDB recommends a 50% size of the GPU memory for AlphaCache.

When the input table data set is much larger (by an order of magnitude) than the size of GPU device memory, a partitioning-based execute strategy has to be used so that the whole device memory is arranged to execute operations on one partition at one time. In this case, AlphaCache should be set to 0 in order to allow a single partition execution to fully utilize all the GPU device memory space. Because even a single query has to sequentially scan multiple partitions, AlphaCache cannot hold the working set to provide effective caching for concurrent query executions.

8 PERFORMANCE EVALUATION

In this section, we present a detailed performance analysis of RateupDB 1.0 version. We first present its pure OLAP performance by executing each of 22 TPC-H queries. Then we present its holistic performance by strictly executing the complete TPC-H benchmark.

System Configuration. We used a Supermicro 743TQ-X11 workstation to conduct all the experiments. The CPU is an Intel Xeon Silver 4215 with a 2.50 GHz frequency, 8 cores (with Hyper-Threading disabled), and 11MB Last-Level Cache. The GPU is an Nvidia Quadro RTX 8000 with 48GB GDDR6 memory and 4,608 CUDA parallel-processing cores [7], which is connected through a PCIe 3.0 bus. The machine has 256GB DDR4 DRAM, and two mirrored 1 TB SSDs. We used Ubuntu 18.04 LTS OS and CUDA Toolkit 10.0.

Workloads and Software. We implemented TPC-H 2.18.0 with three scale factors (1, 10, and 100). To compare RateupDB to the state of the art of commercial GPU database product, we selected the most recent version of OmniSci by considering its wide usage in recent research literatures [44, 90, 99].

8.1 Query Performance

Table 2: Execution times (in seconds) of TPC-H queries by RateupDB and OmniSci. TPC-H scale factors used: 1, 10, and 100. (R: RateupDB, O: OmniSci, DNF: Did Not Finish.)

	R(1)	R(10)	R(100)	O(1)	O(10)	O(100)
Q1	0.10	0.73	7.38	0.28	0.44	2.38
Q2	0.05	0.10	0.33	DNF	DNF	DNF
Q3	0.04	0.17	0.62	0.31	0.82	DNF
Q4	0.02	0.05	0.37	DNF	DNF	DNF
Q5	0.03	0.07	0.87	0.24	0.27	DNF
Q6	0.02	0.08	0.69	0.22	0.23	0.25
Q7	0.04	0.08	0.48	0.23	0.28	DNF
Q8	0.04	0.07	0.40	0.28	0.33	DNF
Q9	0.05	0.19	1.95	0.31	0.34	DNF
Q10	0.13	0.95	1.71	0.67	DNF	DNF
Q11	0.03	0.04	0.18	DNF	DNF	DNF
Q12	0.03	0.08	0.50	0.24	0.25	DNF
Q13	0.03	0.09	1.40	DNF	DNF	DNF
Q14	0.03	0.05	0.27	DNF	DNF	DNF
Q15	0.03	0.09	0.62	0.31	0.40	0.66
Q16	0.06	0.08	0.23	0.97	DNF	DNF
Q17	0.02	0.04	0.39	DNF	DNF	DNF
Q18	0.04	0.24	4.27	0.64	4.82	DNF
Q19	0.03	0.06	0.20	0.23	0.27	0.68
Q20	0.04	0.13	1.35	DNF	DNF	DNF
Q21	0.04	0.13	3.28	DNF	DNF	DNF
Q22	0.03	0.03	0.13	DNF	DNF	DNF

We first measure the read-only query execution performance of RateupDB and OmniSci. Table 2 lists the execution times of all the 22 queries with the three TPC-H scale factors. Each query was executed four times, and the results in the table are the ones at the fourth time execution. In the next several subsections, we look into performance insights into our our observations in experiments.

8.1.1 Overall Analysis. RateupDB can finish all the queries with the three scale factors. However, OmniSci can only finish 13 queries when using scale factor 1. This observation is consistent with a prior study conducted in [44]. It then has more failure cases when increasing to factor 10 and 100. The results show the effectiveness of RateupDB by the subquery optimizations for complex queries .

With scale factor 1, i.e., the total data size is 1GB, RateupDB executed most of queries in less than 0.1s. When increasing to scale factor 10, all query execution times are correspondingly increased, but still less than 1s. OmniSci's performance results are diverse: Some queries' times are almost unchanged from scale factor 1 to 10, for example Q5-Q8; Some queries cannot be finished (Q10, Q16); And some queries have significant long execution times (Q18).

When using scale factor 100, RateupDB has several exceptional queries that cannot be finished in less than 1s. The longest query executions occurred to Q1, Q18, Q21, and Q9, which we will analyze specifically later. For OmniSci, it can only finish 4 out of 22 queries (Q1, Q6, Q15, Q19). By examining the error information, we found most of failed cases are related to three exceptions: (1) *too large hash table*, which is related to multiple joins (e.g., Q9), (2) *too large memory usage* (e.g., Q18), and (3) *type not serializable*, which are often occurred in subqueries handling (e.g., Q2).

8.1.2 Query with heavy aggregation: TPC-H Q1. TPC-H Q1 is a typical query that needs to execute multiple aggregations on a large number of records. RateupDB's execution times for the three scale factors are 0.10s, 0.73s and 7.38s, respectively, which basically reflects a linear time increment with the input data sizes. However, after making careful time breakdowns using a set of micro experiments, we found such an increment is not really caused by the operations like grouping or filtering in Q1, but by the implementation of fixed-point arithmetic operations, which is required for finance-related database workloads (e.g., TPC-H) [85].

Because we cannot use *double precision* to execute TPC-H, GPU's high TFLOPS cannot be easily converted into high performance of fixed-point arithmetic operations. We implement a general-purpose GPU library for decimal representations and calculations, which uses a fixed 20-byte design to support a (36, 30) precision/scale, while OmniSci supports a much smaller precision using a design with 2-8 bytes with the maximized 18 digits for precision. We did not implement any TPC-H-specific optimizations for its decimal type (e.g., [26]) in RateupDB. Therefore its Q1 performance (as well as a part of Q18 that has a subquery with heavy aggregation) is dominated by fixed-point arithmetic operations.

8.1.3 Query with heavy join: TPC-H Q9. As a typical example to OLAP query involving multiple joins, Q9 is often a key performance indicator for database OLAP performance. Due to its optimization for star-join, RateupDB consistently outperforms OmniSci for Q9 by all the three scale factors. RateupDB uses its Star Join operator to execute the multiple joins in the query. In this way, tuples from the fact table can iteratively probe multiple hash tables so that it can achieve the goal of maximized data usage in GPU memory [44, 90]. Although RateupDB does not yet take a JIT compiling-based kernel execution approach, its usage of Star Join operator can effectively make a similar effect. Since a star-like join is a major join pattern in OLAP workloads, RateupDB's implementation can be widely used. In fact, Star Join is also used to execute other TPC-H queries in RateupDB, including Q5, Q7, Q8, and Q10.

8.1.4 Query with complex subquery: TPC-H Q21. Subquery optimization is an important measure for RateupDB to handle various OLAP queries. We also believe that the ability of handling various subqueries is a major leap for a GPU DBMS to become an industry product because academic research prototypes may often focus on simple analytics [41, 90, 115]. As show in Table 2, RateupDB can effectively execute all the TPC-H queries that contain subqueries by using corresponding unnesting techniques.

As a classical case in TPC-H queries, we need to make a significant effort to optimize Q21, which challenges a database system's query optimizer and engine implementation. As introduced in Section 4.3.2, RateupDB uses a Self operator to execute all the subqueries in the query. Reflected by both the Self operator and the Star Join operator used in Q9 (also in Q21), orchestrating multiple operations into a combinational one is a key optimization.

8.2 On Memory Management

We conducted an experiment to examine how two different memory management methods can affect query performance when executing all the 22 TPC-H queries. The first one is RateupDB's old solution only using ManagedMemory, and the second one is RateupDB's new solution with AlphaCache. In this experiment, AlphaCache takes 50% of GPU device memory, i.e., 24GB. Figure 4 shows the speedups of the new solution over the old one for all the 22 queries, which shows that the execution of most of queries are significantly accelerated, and the average speedup is 1.22X.

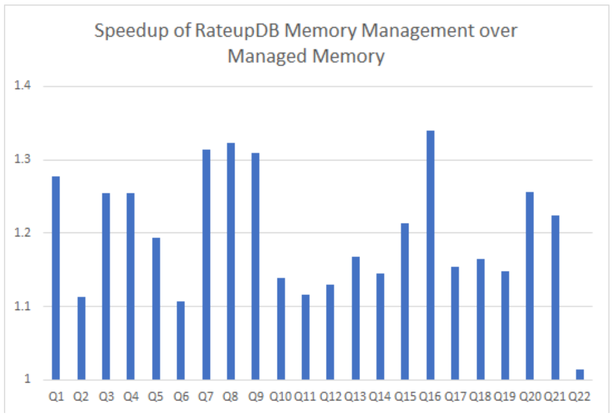

Figure 4: Performance comparison between RateupDB's hybrid memory management solution and default ManagedMemory-based solution.

With ManagedMemory, the query processing on both intermediate data and the input table data are automatically managed by the GPU VM mechanism. However, the disadvantage of this approach is that the database cannot control the memory space contention between these two kinds of data. With the new solution, we explicitly divide the memory space into the table zone (AlphaCache) to cache the shared table data and the work zone to store non-shared query intermediate data. Therefore, we can guarantee that the intermediate data are kept in GPU device memory during query execution and cache replacement can only occur among input table data.

8.3 Transaction Performance

TPC-H requires two refresh functions (RF1 and RF2) to execute concurrent inserts and deletes to *ORDERS* and *LINEITEM*. We have implemented the two functions using two Snapshot Isolation transactions in RateupDB. The new inserted data items and the keys for deleted data items are first stored in several temporary tables, which are then used by the transactions. The code sections of RF1 and RF2 illustrates their logic flows, where $t1 - t4$ are the temporary tables. For TPC-H scale factor 100, the number of rows being inserted or deleted is 150,000 for *ORDERS* and (1-7) times more for *LINEITEM*. We use primary keys for the two tables but do not use any foreign key connecting the two tables. This is allowable by the TPC-H specification. When executing RF1, the insert will execute additional constraint check for primary key conflicts.

```
RF1:
begin transaction;
insert into ORDERS select * from t1;
insert into LINEITEM select * from t2;
commit;
RF2:
begin transaction;
delete from LINEITEM
    where L_ORDERKEY in (select orderkey from t3);
delete from ORDERS
    where O_ORDERKEY in (select orderkey from t4);
commit;
```

According to OmniSci's document, transactions are not supported. So we cannot compare its performance with RateupDB for this workload. Furthermore, there is no public TPC-H results for any GPU-accelerated database products. Therefore we select published performance results of CPU databases for performance comparisons. Since RateupDB is a non-cluster system, we choose the fastest non-cluster result on TPC website for the same scale factor (Actian VectorWise 3.0.0). Table 3 lists the execution times of RF1 and RF2 by the two systems for the TPC-H Power Test, in which the transactions are executed solely without any interference. The results show that the execution times are comparable. However, the implementation of RF1 and RF2 in the two systems are not necessarily the same, considering multiple factors such as index building, constrain checking, and transaction implementations.

Table 3: Execution times (s) of TPC-H RF1 and RF2 by RateupDB and VectorWise for TPC-H Power Test.

	RateupDB 1.0	VectorWise 3.0.0
RF1	4.06	6.5
RF2	1.71	2.1

8.4 RateupDB for Complete TPC-H Test

After presenting the OLAP performance and the transaction performance, we are now in a position to examine how RateupDB can handle hybrid workloads with both OLAP and OLTP operations. We use a complete, industry-standard TPC-H benchmark execution as the workload, which has three distinct features compared to the commonly used performance characterization workload in various research literature. First, its power test part applies a strict order of sequential executions of the 22 queries. Second, its throughput test part requires explicit transaction execution streams along with query execution streams in order to examine the effects of co-running OLAP and OLTP. Third, it requires a minimum streams

Table 4: Complete TPC-H results compared to representative published results (TPC-H Scale Factor 100).

Database	Hardware	Power	Throughput	Composite	Report Date
EXASOL EXASolution 5.0	6-node cluster, 120 cores	1,265,179.1	1,980,000.0	1,582,736.4	09/23/14
Actian VectorWise 3.0.0	non-cluster, 16 cores	458,664.5	384,764.9	420,092.4	05/13/13
RateupDB 1.0	non-cluster, 8 cores, RTX8000	354,183.3	296,476.0	324,047.6	Not Yet
Goldilocks v3.1	6-node cluster, 216 cores	6,569.7	87,954.7	243,07.0	12/19/18

of concurrent OLAP query so that the query engine must balance both the overall throughout and single query response time.

Table 4 shows RateupDB's complete TPC-H performance compared to the published results of three other database systems, which are publicly available on the TPC website. We choose (1) the fastest one (EXASOL EXASolution 5.0) regardless hardware configurations, (2) the fastest single-node non-cluster system (Actian VectorWise 3.0.0), and (3) the most recently published one for the scale (Goldilocks v3.1). Compared to the three CPU-only database systems, RateupDB currently is in a position between the highly-optimized analytical database (i.e., VectorWise) and the hybrid database (i.e., Goldilocks). Detailed analysis is out of the scope of the paper due to the page limit. RateupDB's TPC-H performance in Table 4 is limited by its unoptimized fixed-point arithmetic implementation and its usage of a single GPU device.

9 AN EXCHANGE OF OPPOSITE VIEWS

We follow a critical thinking style by discussing *the fallacy and the pitfall* [54, 60], which are based on the lessons and insights about issues that matter from our experiences of building RateupDB.

9.1 Fallacies

Fallacy (1): An algorithm's adoption in products only depends on its performance. In practice, the selection of an algorithm is underlying systems dependent and development cost dependent. Specifically, we must consider multiple factors, including its execution performance, its adaptiveness to different dynamic situations (e.g., data distributions and hardware parameters), implementation easiness and the decoupling degree to other system components. Another challenge is to develop a clear and standing-the-test rule to judge under what conditions the algorithm should be used.
Fallacy (2): A HTAP DBMS only needs a unified hybrid data format. Despite the existence of various unified hybrid data formats that combine the advantages for both row store and column store, the rise of Heterogeneous HTAP (H^2TAP) DBMS brings new challenges to data store with a set of new requirements including hardware efficiency, performance isolation, and engineering cost. As shown by a set of industry-level HTAP database products, a partitioned, multi-staged, and combined data store can effectively support hybrid workloads, particularly in GPU-accelerated systems.
Fallacy (3): A GPU DBMS's performance depends only on how to execute database operations on GPU. Existing research-oriented GPU database projects are often focused on how to best utilize GPUs to accelerate certain database operations. However, significant performance improvements are often made at a logic level of the database, which are independent of the GPU implementations. TPC-H Q21 is such an example, which demonstrates the benefit of best utilizing the intra-query correlation information.

9.2 Pitfalls

Pitfall (1): Determining when to use an algorithm is even harder than how to implement it. The role of query optimization has been studied for decades, but unsolved issues still exist now. There are multiple factors to be considered in an unified cost function to estimate the execution cost of a physical operator. However, achieving this goal is a non-trivial task due to the existence of different dynamic factors. RateupDB's experience tends to avoid this problem (if possible) instead of directly solving it.
Pitfall (2): GPU query performance is closely related to the underlying data type. GPU data type implementation must consider storage overhead and computing efficiency. Performance of CUDA primitive data types and user-defined complex data types are very different, for example the decimal type, which requires fixed-point arithmetic operations instead of GPU-inherent floating point support. Comparing GPU performance without considering data type implementations can make misleading conclusions.
Pitfall (3): A GPU query engine cannot only rely on wrapping various CUDA libraries. The rapid development of Nvidia CUDA ecosystem has provided effective functions, such as the Unified Memory APIs and other library functions (e.g., CUB or Thrust). However, these CUDA libraries cannot fully satisfy key requirements of a complete database, for example, handling arbitrary data sizes and types. A GPU query engine should have its own solutions without solely relying on CUDA libraries.
Pitfall (4): Advanced GPU's VM facilities are not sufficient to manage device memory locality. The separation between the device memory and the host memory creates a challenge of GPU memory management. A query engine only using GPU's VM facilities can have uncontrolled performance degradations. RateupDB's solution is to manually manage device memory space for both data locality and query performance.

10 CONCLUSION

We present a comprehensive study of RateupDB, a high performance database system for both OLAP and OLTP workloads by CPU/GPU. A major contribution of this paper is to have identified a large spectrum of design possibilities for RateupDB, aiming to justify the art of balance in its design and implementation. Based on our product development experiences, we provide a set of critical discussions on multiple fallacies and pitfalls. To the best of our knowledge, this is the first paper to systematically explain a GPU HTAP system on its design and implementation, and its holistic performance by the industry-standard database benchmark (TPC-H). We believe that our experience of building and evaluating RateupDB would benefit the research and development of high performance databases for both academia and industries.

REFERENCES

[1] [n.d.]. *BigQuery ML.* https://cloud.google.com/bigquery-ml/docs/introduction
[2] [n.d.]. *Brytlyt.* https://www.brytlyt.com/
[3] [n.d.]. *CUB.* https://nvlabs.github.io/cub/
[4] [n.d.]. *HeteroDB.* https://www.heterodb.com
[5] [n.d.]. *Introducing AresDB: Uber's GPU-Powered Open Source, Real-time Analytics Engine.* https://eng.uber.com/aresdb/
[6] [n.d.]. *Kinetica.* https://www.kinetica.com/
[7] [n.d.]. *Nvidia Quadro RTX 8000.* https://www.nvidia.com/en-us/design-visualization/quadro/rtx-8000/
[8] [n.d.]. *NVLINK AND NVSWITCH.* https://www.nvidia.com/en-us/data-center/nvlink/
[9] [n.d.]. *OmniSci.* https://www.omnisci.com/
[10] [n.d.]. *SQream.* https://www.sqream.com/
[11] [n.d.]. *TPC-H.* http://tpc.org/tpch/default5.asp
[12] [n.d.]. *USPS Deploys Kinetica to Optimize its Business Operations.* https://www.kinetica.com/wp-content/uploads/2016/12/Kinetica_CaseStudy_USPS_1.0_WEB.pdf
[13] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* 967–980.
[14] Athena Ahmadi, Alexander Behm, Nagesh Honnalli, Chen Li, Lingjie Weng, and Xiaohui Xie. 2012. Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic acids research* 40, 6 (2012), e41–e41.
[15] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.
[16] Ablimit Aji, George Teodoro, and Fusheng Wang. 2014. Haggis: turbocharge a MapReduce based spatial data warehousing system with GPU engine. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data.* 15–20.
[17] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proc. VLDB Endow.* 5, 10 (June 2012), 1064–1075.
[18] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research.*
[19] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture.. In *EDBT*, Vol. 2016. 461–466.
[20] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* 136–150.
[21] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.
[22] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2004. 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.* 1021–1032.
[23] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. 2009. Enhanced subquery optimizations in oracle. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1366–1377.
[24] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.
[25] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data.* 37–48.
[26] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking.* Springer, 61–76.
[27] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, Vol. 99. 54–65.
[28] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209.
[29] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20).* 209–223.
[30] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324.
[31] Steven WD Chien, Ivy B Peng, and Stefano Markidis. 2019. Performance evaluation of advanced features in CUDA unified memory. *arXiv preprint arXiv:1910.09598* (2019).
[32] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. 2020. Buddy compression: Enabling larger memory for deep learning and HPC workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 926–939.
[33] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. 2009. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology.* 1–11.
[34] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.
[35] Harish Doraiswamy and Juliana Freire. 2020. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1875–1885.
[36] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.
[37] Mostafa Elhemali, César A Galindo-Legaria, Torsten Grabs, and Milind M Joshi. 2007. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data.* 993–1004.
[38] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 670–680.
[39] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.
[40] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.
[41] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. Nestgpu: Nested query processing on gpu. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 1008–1019.
[42] Phil Francisco. 2011. IBM PureData System for Analytics Architecture A Platform for High Performance Data Warehousing and Analytics. In *https://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.*
[43] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data.* 1603–1618.
[44] Henning Funke and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *Proceedings of the VLDB Endowment* 13, 6 (2020), 884–897.
[45] Oded Green, Robert McColl, and David A Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing.* 331–340.
[46] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abaigail Hsu, Hector Carrillo Carrillo, Hessoo Kim, et al. 2018. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* IEEE, 24–36.
[47] Mark Harris. 2017. Unified Memory for CUDA Beginners. In *https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.*
[48] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–39.
[49] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 511–524.
[50] Bingsheng He and Jeffrey Xu Yu. 2011. High-Throughput Transaction Executions on Graphics Processors. *Proceedings of the VLDB Endowment* 4, 5 (2011).
[51] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering.* IEEE, 1199–1208.
[52] Max Heimel and Volker Markl. 2012. A First Step Towards GPU-assisted Query Optimization. *ADMS@ VLDB* 2012 (2012), 33–44.
[53] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment* 6, 9 (2013), 709–720.
[54] John L Hennessy and David A Patterson. 2018. *Computer architecture: a quantitative approach, 6th edition.* Elsevier.
[55] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–37.
[56] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing.* 43–57.

[57] 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 *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1235–1246.

[58] Yin Huai, Siyuan Ma, Rubao Lee, Owen O'Malley, and Xiaodong Zhang. 2013. Understanding insights into the basic structure and essential issues of table placement methods in clusters. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1750–1761.

[59] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[60] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[61] Tomas Karnagel, René Müller, and Guy M Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@ VLDB* 8 (2015), 20.

[62] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[63] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[64] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[65] Won Kim. 1982. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems (TODS)* 7, 3 (1982), 443–469.

[66] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[67] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. In *CIDR*.

[68] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[69] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[70] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[71] Choudur Lakshminarayan, Thiagarajan Ramakrishnan, Awny Al-Omari, Khaled Bouaziz, Faraz Ahmad, Sri Raghavan, and Prama Agarwal. 2019. Enterprise-wide Machine Learning using Teradata Vantage: An Integrated Analytics Platform. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2043–2046.

[72] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[73] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2014. VAST: The illusion of a large memory space for GPUs. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 443–454.

[74] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. 2009. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. *Proceedings of the VLDB Endowment* 2, 1 (2009), 373–384.

[75] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. 2011. Ysmart: Yet another sql-to-mapreduce translator. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 25–36.

[76] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment* 13, 3 (2019), 348–361.

[77] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[78] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[79] Yanhui Liang, Hoang Vo, Ablimit Aji, Jun Kong, and Fusheng Wang. 2016. Scalable 3d spatial queries for analytical pathology imaging with mapreduce. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 1–4.

[80] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. 2012. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics* 28, 6 (2012), 878–879.

[81] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 37–50.

[82] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul23. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[83] Andreas Meister. 2015. GPU-accelerated join-order optimization. In *The VLDB PhD workshop, PVLDB*, Vol. 176. 1.

[84] Todd Mostak. 2013. An overview of MapD (massively parallel database). *White paper. Massachusetts Institute of Technology* (2013).

[85] Thomas Neumann. 2015. The price of correctness. In *http://databasearchitects.blogspot.com/2015/12/the-price-of-correctness.html*.

[86] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[87] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. 2012. Merge path-parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1611–1618.

[88] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1099–1110.

[89] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.

[90] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment* 14, 2 (2020), 202–214.

[91] Sukhada Pendse, Vasudha Krishnaswamy, Kartik Kulkarni, Yunrui Li, Tirthankar Lahiri, Vivekanandhan Raja, Jing Zheng, Mahesh Girkar, and Akshay Kulkarni. 2020. Oracle Database In-Memory on Active Data Guard: Real-time Analytics on a Standby Database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1570–1578.

[92] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2016. A metric for performance portability. *arXiv preprint arXiv:1611.07409* (2016).

[93] An Qin, Mengbai Xiao, Jin Ma, Dai Tan, Rubao Lee, and Xiaodong Zhang. 2019. DirectLoad: A Fast Web-scale Index System across Large Regional Centers. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1790–1801.

[94] An Qin, Yuan Yuan, Dai Tan, Pengyu Sun, Xiang Zhang, Hao Cao, Rubao Lee, and Xiaodong Zhang. 2017. Feisu: Fast Query Execution over Heterogeneous Data Sources on Large-Scale Clusters. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1173–1182.

[95] 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 *Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[96] Lucas C Villa Real and Bruno Silva. 2018. Full Speed Ahead: 3D Spatial Database Acceleration with GPUs. *Proceedings of the VLDB Endowment* 11, 9 (2018).

[97] Ran Rui, Hao Li, and Yi-Cheng Tu. 2021. Eficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. *Proceedings of the VLDB Endowment* 14 (2021).

[98] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th international conference on scientific and statistical database management*. 1–12.

[99] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[100] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.

[101] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 698–709.

[102] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 417–432.

[103] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2005. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*. 553–564.

[104] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.

[105] Diego G Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A Boncz. 2018. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing.. In *ADMS@ VLDB*. 1–10.

[106] Luan Tran, Min Y Mun, and Cyrus Shahabi. 2020. Real-time distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment* 14, 2 (2020), 141–153.

[107] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1153–1170.

[108] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[109] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: Device memory management for GPGPU computing. *ACM SIGMETRICS Performance Evaluation Review* 42, 1 (2014), 533–545.

[110] Kaibo Wang, Yin Huai1 Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. 2012. Accelerating Pathology Image Data Cross-Comparison on CPU-GPU Hybrid Systems. *Proceedings of the VLDB Endowment* 5, 11 (2012).

[111] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.

[112] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record* 45, 2 (2016), 17–22.

[113] 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.

[114] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

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

[116] 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 *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 273–283.

[117] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.

[118] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 10–10.

[119] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. Analyticdb: Real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[120] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

[121] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[122] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.