A Comprehensive Overview of GPU Accelerated Databases

Devesh Sarda
University of Wisconsin-Madison

Harshit Sharma
University of Wisconsin-Madison

Anmol Sharma
University of Wisconsin-Madison

Vaibhav Nitnaware
University of Wisconsin-Madison

Abstract

Over the past decade, the landscape of data analytics has seen a notable shift towards heterogeneous architectures, particularly the integration of GPUs to enhance overall performance. In the realm of in-memory analytics, which often grapples with memory bandwidth constraints, the adoption of GPUs has proven advantageous, thanks to their superior bandwidth capabilities. The parallel processing prowess of GPUs stands out, providing exceptional efficiency for data-intensive workloads and outpacing traditional CPUs in terms of data processing speed. While GPU databases capitalize on these strengths, there remains a scarcity of comparative studies across different GPU systems. In light of this emerging interest in GPU databases for data analytics, this paper proposes a survey encompassing multiple GPU database systems. The focus will be on elucidating the underlying mechanisms employed to deliver results and key performance metrics, utilizing benchmarks such as SSB and TPCH. This undertaking aims to shed light on new avenues for research within the realm of GPU databases.

1 Introduction

Over the last decade, the landscape of data analytics has evolved significantly, with a notable shift toward embracing heterogeneous architectures, such as the integration of Graphics Processing Units (GPUs), CXL, and SmartNIC to bolster performance. In this study, we will focus mainly on GPUs. In-memory analytics, a domain often constrained by memory bandwidth limitations, has particularly benefited from the enhanced bandwidth capacities of GPUs, thereby motivating their widespread adoption in analytical query processing. GPUs, renowned for their innate parallel processing capabilities, have proven highly adept at managing data-intensive workloads, delivering notably swifter data processing in comparison to conventional Central Processing Units (CPUs).

Despite the widespread integration of GPUs in database systems, a notable gap exists in the domain of comprehensive comparative analyses among different GPU systems. Addressing this gap, this paper proposes a comprehensive investigation into multiple GPU database systems. The focus will be on crucial performance metrics such as Query Execution Time and a detailed exploration of the underlying architecture of these GPU systems. The primary aim of this study is to survey diverse GPU database systems, providing valuable insights into their evolving potential and identifying avenues for improvement.

2 Previous Works

Most current open source systems [2, 5, 6] utilize a hybrid approach when it comes to GPU databases in that they perform most of their computation on the CPU and offload selected computation onto the GPU. Other systems [4] first transform the data into other formats, such as PyTorch tensors, and perform kernel operations on them, incurring a high overhead. Purely research systems [15] that are GPU only support a limited number of operations due to one primary reason: A limited amount of data can be stored on a GPU. Some previous solutions to challenge have been (1) using compression to fit more data on the GPU [17], (2) utilizing multiple GPUs [14].

Previous work [3] have looked into benchmarking such systems but have primarily utilized SSB schema [11] as a benchmark. However, the SSB schema is a simplified version of the general TPC-H[12] schema which contains more tables and a wider range of operations in its queries. Additionally, while previous work focuses on metrics such as execution time, which are important, they often fail to capture metrics related to the deployment of such systems in a production environment. Finally, most metrics reported by these systems are based on a performance on a singular system and never address how well these translate to other hardware configurations.

3 Methodology

The approach to conducting a comprehensive survey and performance comparison of GPU database systems involved a well-defined methodology. The initial step was the careful selection of GPU database systems with an emphasis on diversity in features and use cases. BlazingSQL[2], HeavyDB[5], TQP[4], and Crystal[15] were identified as the most suitable candidates for this comparative study. Execution time was established as the primary performance metric in the subsequent critical phase.

To ensure a thorough assessment, existing benchmark studies from prior research were examined, covering various scale factors against different benchmarks. Subsequently, experiments were conducted to obtain benchmark results for the selected systems. Notably, Crystal presented a challenge as it lacked support for TPCH queries, prompting additional efforts to address this limitation. Moreover, to fill a gap in previous work, a CPU-only OLAP DBMS comparison was introduced using DuckDB as the benchmark tool. Data preparation involved creating representative datasets with scale factors up to 16 for SSB and up to 8 for TPCH.

The benchmarking workloads were strategically designed to encompass a spectrum of query complexities and data access patterns, facilitating a holistic evaluation of system performance. In the final phase, the findings from the experiments were synthesized, shedding light on both quantitative and qualitative differences between the GPU database systems under investigation. This systematic methodology provided valuable insights into the strengths and potential areas of improvement for each system.

4 Survey of Database Systems

Understanding various GPU database systems is crucial for gaining insights into the diverse ways these systems handle workloads and queries. The following section is dedicated to exploring different GPU databases that are widely used, aiming to comprehend the distinct architectures employed in their GPU database designs. The databases discussed in this section include:

- **BlazingSQL**
- **OmniSciDB**
- **Crystal+**
- **Tensor Query Processor (TQP)**

Each of these GPU databases represents a unique approach to leveraging GPU technology for efficient data processing. By delving into their architectures and functionalities, one can gain a comprehensive understanding of the landscape of GPU-accelerated database systems and their respective contributions to handling varied workloads and queries.

4.1 BlazingSQL

4.1.1. Introduction:

BlazingSQL [2] functions as a robust SQL interface tailored for cuDF, offering advanced features to enhance data science workflows and manage large-scale enterprise datasets. Its notable integration with the dask-cudf library, part of the broader RAPIDS project, enhances adaptability and resilience, making BlazingSQL adept at addressing diverse data science tasks.

The adaptability and versatility of BlazingSQL are evident in its support for various formats and frameworks, positioning it as a crucial component in the data science ecosystem. Providing an efficient pathway from raw data handling to advanced analytics and machine learning, BlazingSQL streamlines the entire data science workflow, offering a cohesive solution. The open accessibility of BlazingSQL's foundational code, released under the Apache 2.0 License, reflects a commitment to transparency and community collaboration, fostering an open environment for development and innovation.

4.1.2. Architecture and Features:

Upon reviewing Figure 1, it becomes evident that BlazingSQL initiates a connection with Apache Calcite via JPype. This connection serves a crucial role by utilizing Apache Calcite as a SQL parser, employing its capabilities to translate SQL strings into a relational algebra plan. The Relational Algebra Engine (RAL) plays a pivotal role in this intricate process, taking charge of generating a distributed homogeneous execution graph. This graph serves as a key instrument, communicating essential processing responsibilities to each worker within the system, thereby orchestrating an efficient and coordinated execution of the query.

The relational algebra in BlazingSQL transforms into a physical relational algebra plan, forming a directed acyclic graph (DAG) where nodes represent kernels and edges denote caches. Kernels logically organize transformations on distributed `DataFrames`, and `CacheData` objects, which can be in various states like `GPUCacheData` or `CPUCacheData`, facilitate data storage without immediate materialization. Most kernels utilize CacheData from a CacheMachine to create tasks for the task executor. The task executor, exclusive to

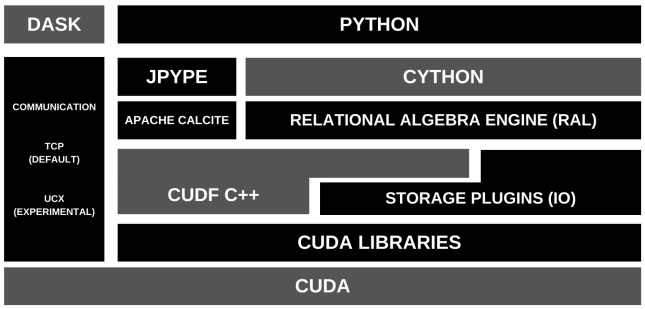


Figure 1: BlazingSQL RAL Architecture

Nvidia GPUs, manages resource access, handles memory, and supports retries for operations initially hindered by resource scarcity.

Upon residing as GPU `DataFrames` in GPU memory, users can leverage `RAPIDS cuML` for diverse machine learning applications or convert them to formats like `DLPack` or `NVTabular`, enabling in-GPU deep learning with frameworks like PyTorch or TensorFlow. In practical terms, BlazingSQL excels in Extract, Transform, Load (ETL) processes, efficiently transferring raw data straight into GPU memory, and transforming it into optimized GPU `DataFrames` for further analytical tasks.

BlazingSQL's proficiency extends to querying externally stored data, simplifying the process with standard SQL commands. The results manifest as `GPU DataFrames` (GDFs) in GPU memory, seamlessly integrating with RAPIDS libraries for diverse data science workloads. This capability enhances BlazingSQL's efficiency in integrating external data sources into GPU-accelerated analytics and processing pipelines

4.1.3. Example of End-to-End Query Execution:

Consider the example given below:

```
Qx: SELECT o_custkey, SUM(o_totalprice)
    FROM orders
    WHERE o_orderkey < 10
    GROUP BY o_custkey;
```

In the provided example query (`Qx`), when executed in the BlazingSQL Core engine, the SQL query undergoes parsing and optimization by Apache Calcite. The resulting optimized algebra, along with data sources like cudfs or files, is distributed to workers via `Dask`. The Relational Algebra (RAL) representation is as follows:

```
LogicalProject(o_custkey=[$0],
EXPR$1=[CASE(=($2, 0), null:DOUBLE, $1)])
  LogicalAggregate(group=[{0}], EXPR$1=[$SUM0($1)],
  agg#1=[COUNT($1)])
    LogicalProject(o_custkey=[$1], o_totalprice=[$2])
      BindableTableScan(
      table=[[main, orders]], filters=[[<($0, 10)]],
      projects=[[0, 1, 3]],
      aliases=[[$f0, o_custkey, o_totalprice]])
```

On each worker, the relational algebra is translated into a physical plan, exemplified below, where each relational algebra step corresponds to one or more physical plan steps. This physical plan constructs an execution graph forming a Directed Acyclic Graph

(DAG) of kernels and caches. The cache's purpose is to store data as `CacheData` between computational stages, enabling data movement across different memory layers to scale beyond the capacity of a single layer.

```
LogicalProject(o_custkey=[$0],
EXPR$1=[CASE(=($2, 0), null:DOUBLE, $1)])
  MergeAggregate(group=[{0}],
  EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])
    ComputeAggregate(group=[{0}],
    EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])
      LogicalProject(o_custkey=[$1], o_totalprice=[$2])
        BindableTableScan(table=[[main, orders]],
        filters=[[<($0, 10)]], projects=[[0, 1, 3]],
        aliases=[[$f0, o_custkey, o_totalprice]])
```

Within the DAG, kernels are exclusively connected through caches. These kernels orchestrate complex distributed operations, generating tasks sent to the Task Executor for execution. The DAG's final output is a Cache containing the result.

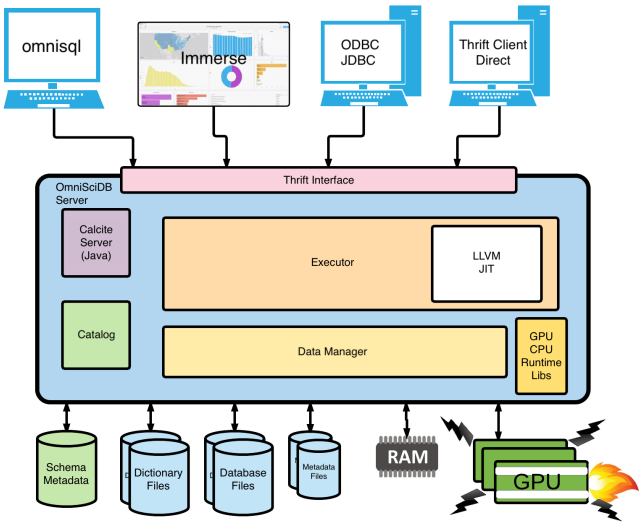


Figure 2: OmniSciDB High-Level Architecture

4.2 OmniSciDB

4.2.1. Introduction:

OmniSciDB or HeavyDB is a state-of-the-art database that specializes in real-time analytics on large data sets through GPU acceleration. This cutting-edge platform is designed to optimize data processing, utilizing the raw power of GPUs to outperform traditional CPU-based databases. With its in-memory, columnar storage, and native SQL compatibility, HeavyDB ensures fast, efficient analytics operations.

A key feature of HeavyDB is its built-in visualization capabilities, enabling instant graphical data analysis and eliminating the need for separate visualization tools. Known for speed, scalability, and adaptability, HeavyDB caters to various sectors, including finance and telecommunications, where rapid data analysis is essential.

Despite its reliance on specific hardware, HeavyDB's ability to integrate with common data science environments and scale with additional GPU resources makes it a versatile and powerful tool in the realm of big data and analytics. As a leader in GPU-accelerated data processing, HeavyDB is pivotal for businesses aiming to achieve swift and insightful data-driven decisions.

4.2.2. Architecture:

OmniSciDB distinguishes itself with a GPU-centric processing approach, ensuring each operation is finely tuned for GPU efficiency. At the heart of its architecture as shown in Figure 2, lies Apache Thrift, which standardizes communication for both external clients and internal processes, facilitating seamless interactions with various client tools, including the command line interface omnisql, JDBC driver, and `SQLImporter` utility.

For query optimization, it leverages Apache Calcite, renowned for its modular nature and flexibility. This enables the addition of custom functions to the SQL parsing process, such as trigonometric computations vital for geospatial analysis, ensuring seamless integration and optimization of these functions within the query plans.

The database's structure is further strengthened by its Catalog component, which manages metadata through a centralized system. Each database maintains its own Catalog, all orchestrated under a System Catalog that actively manages metadata repositories via an SQLite database. This centralized metadata management exemplifies OmniSciDB's layered approach to organizing and retrieving data efficiently.

Adding to these features, it incorporates an integrated visualization engine, directly transforming data into visual analytics and enhancing the data-to-insight transition. Its robust SQL interface provides a familiar landscape for traditional database users, making the shift to this powerful GPU-driven analytics platform smoother. The system's advanced memory management and cross-filtering functionalities significantly bolster its data processing capabilities, while its columnar data storage format ensures fast read operations and optimal data compression.

Through this integration of sophisticated components, it emerges as a highly efficient and scalable solution, capable of handling diverse analytical workloads and leading the way in the realm of GPU-accelerated data analytics.

4.2.3. Execution Model of Query:

The execution model in Figure 3 is the sophisticated process a GPU-accelerated database like OmniSciDB uses to handle SQL queries. It begins with parsing and validating SQL, then moves on to create an optimized relational algebra sequence. The system prepares for execution by loading necessary data and respecting data ownership and identification protocols.

Query kernels are then executed on the target device—GPU or CPU—leveraging the parallel processing power of GPUs for speed. After execution, results are consolidated, and the system checks for query completion. If incomplete, it cycles through execution as needed. Once finished, results are returned, showcasing OmniSciDB's efficient handling of complex, data-intensive queries for real-time analytics.

4.2.4. Advantages and Industry Application:

The real-time processing and visualization capabilities of HeavyDB

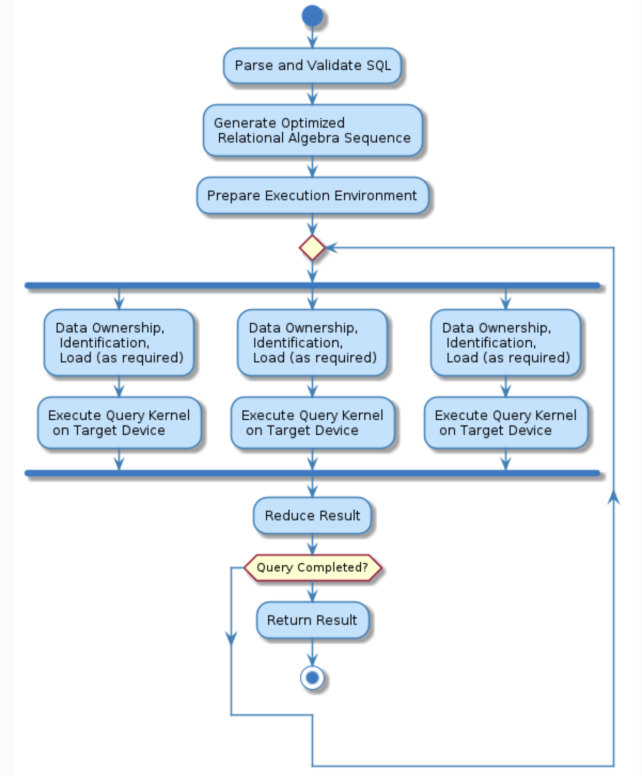


Figure 3: Query Execution in OmniSciDB

make it invaluable across various sectors, including financial services, telecommunications, retail, and public safety. Its GPU-first approach provides speed and performance, with scalability achieved through horizontal GPU integration. Although requiring specific GPU hardware presents a limitation, HeavyDB's ability to integrate with data science tools like Python and R compensates by adding versatility. As data volumes grow and the demand for quick analytics increases, HeavyDB's focus on GPU utilization and real-time data handling solidifies its position as a key player in data management and analysis.

4.3 Crystal+

4.3.1. Introduction:

Crystal+ is a collection of block-wide device functions that can be used to implement high-level SQL queries based on the original Crystal library [16]. In this section, we will cover the original Crystal library and then dive into the improvements made in the next section. The Crystal library stores the table in columnar format as well as perform dictionary encoding [1] to convert all non-numerical data to a numerical format. Crystal stores the working set of the data we are operating on in the GPU memory itself rather than using a coprocessing model to move the data to the GPU during query time. The benefit that Crytal has over the other data in GPU solutions is that it uses a tile-based execution model. Specifically, the elements we want to work on are broken into tiles and each thread block is responsible for one tile. This also involves loading the data into shared memory of the GPU once and then performing all operations in shared memory thus limiting the I/O bandwidth to fetch and write the data. It then implements a series of device functions such as `BlockAggergate` that utilize the tile-based execution model and then combine these blocks to implement various SSB queries. Note that these atomics are relatively limited and can't be combined to implement all queries, such as the TPC-H benchmark. Additionally, all of the Crystal implementation relies on internal data structures and doesn't take advantage of the advancements made by NVIDIA in the CUDA programming framework.

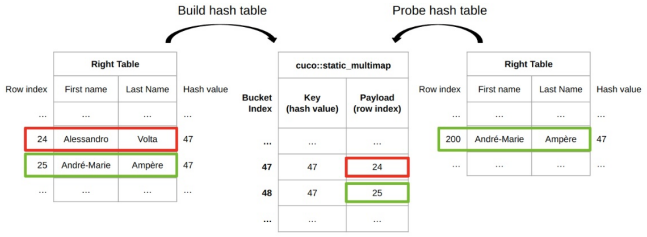


Figure 4: How Crystal+ implements hash-based join

4.3.2. Improvements made:

As mentioned above, the original Crystal doesn't take advantage of the advancements in GPU programming. In order to do so, the original blocks implemented in the Crystal repo were reimplemented using the Thrust [10], Cub [8], and cuCollections [9] libraries. Not only does this give better performance than the naive array-based implementations as we increase the size of the dataset but also is more modular and adaptable to take advantage of the improvements made in different NVIDIA hardware.

One of the main operations that is performed in most analytics SQL queries is joins and we can take advantage of the cuCollections library to implement an efficient hash-based join as illustrated in Figure 4. This involves two different phases of build phase and probe hash similar to the implementation of hash-based join on the CPU. However, we are going to be using cuCollection's multi-map to store the data and thus each thread in the kernel in the build and probe phase just needs to be responsible for determining the row's key based on the row's values.

One another common operation that is performed in many queries is multi-column group by and thus Crystal+ implements an efficient group by using the static map. It does so by determining the maximum value for each column and using that to compute the number of bits necessary to represent all of the values in the column which we will call width. It then uses the width to determine a bitmask as well as a key offset for each column. Then for each row, it first applies the bit mask for each of the column's values and then performs the bit shift for that value and then combines the values to generate a group key. It then performs the associated aggregation for that group and each thread does it for the row it is responsible for. Once the kernel has finished execution, we can get the aggregate for each group as well as use the bit mask and bit shift to extract the values that made up the group.

4.3.3. *Query Execution:*

In order to determine the block functions we need to combine for a given query, we utilize Postgres's `EXPLAIN` [13] functionality to generate a query plan. Consider the example query:

```
SELECT l_returnflag, l_linestatus, COUNT(*),
FROM lineitem, orders,
WHERE lineitem.orderkey = orders.orderkey,
GROUP BY l_returnflag, l_linestatus;
```

Running `EXPLAIN` in Postgres for this query gives us a query plan of:

Finalize GroupAggregate
Group Key: lineitem.l_returnflag, lineitem.l_linestatus
→ Gather Merge
Workers Planned: 2
→ Sort
Sort Key: lineitem.l_returnflag, lineitem.l_linestatus
→ Partial HashAggregate
Group Key: lineitem.l_returnflag, lineitem.l_linestatus
→ Parallel Hash Join
Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
→ Parallel Seq Scan on lineitem
→ Parallel Hash
→ Parallel Scan using orders

Anytime the query plan mentions a join, we utilize the hash joined mentioned the above section and when the query plan mentions an aggregate we utilize the aggregate mentioned in the section above.

4.4 Tensor Query Processor (TQP)

4.4.1. *Introduction:*

Tensor Query Processor (TQP) [4] represents a novel paradigm in analytical database management systems, aiming to leverage the strengths of both relational databases and tensor computing within a unified framework. TQP transforms traditional SQL queries into tensor programs and executes them on TCRs such as PyTorch, TensorFlow, TVM, ONNX, etc. TQP introduces a set of novel algorithms and a compiler stack for converting relational operators into tensor computations. This helps TQP achieve the following three goals:

- **Performance:** Deliver significant performance improvements over CPU-based data systems, and match or outperform custom-built solutions for GPUs. TQP capitalizes on the computational prowess of TCRs and provides a TCR-aware query optimizer to extract the best performance from the underlying hardware.
- **Portability:** Demonstrate portability across a wide range of target hardware and software platforms. Being hardware agnostic allows TQP to adapt its execution strategies to ensure optimal performance across diverse computing environments ranging from CPUs, to discrete GPUs, integrated GPUs (Intel and AMD), NN-accelerators (TPUs), and web browsers.

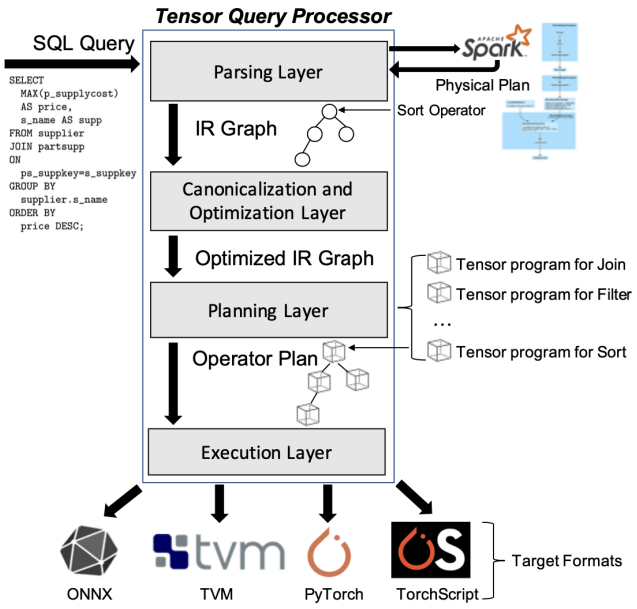


Figure 5: TQP's compilation phase

- **Parsimony:** Prioritize developer productivity by providing a robust, flexible and sustainable framework. TQP's sophisticated compiler stack automates the translation of SQL to optimized tensor programs, reducing the need for manual coding and intricate optimization techniques.

4.4.2. *Architecture and Features:*

TQP operates in two phases:

(1) **Compilation:** Figure 5 illustrates the four-step transformation of an SQL query into an executable tensor program.
   - Parsing Layer: Constructs an internal *intermediate representation (IR)* graph depicting the query's physical plan.
   - Optimization Layer: Performs canonicalization and applies optimization rules to yield an *optimized IR graph*.
   - Planning Layer: Translates the optimized IR graph into an *operator plan* containing a mapping of each operator to a tensor program implementation.
   - Execution Layer: Generates an *executor* object responsible for orchestrating the tensor program execution. It sequentially processes data tensors while managing memory through garbage collection. Additionally, it supports dynamic compilation for various target formats (e.g., PyTorch, TVM, ONNX).

(2) **Execution:** This phase involves transforming the input data into tensors (refer Figure 6 and feeding it into the executor object, returning the query result in tensor, NumPy, or Pandas format. TQP exploits the tensor-level intra-operator parallelism provided by the TCRs.

The code sample below demonstrates submitting a query string to the TQP compiler and then running the compiled query object generating the output in Pandas dataframe format.

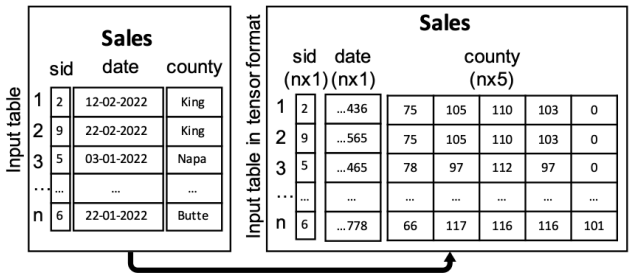


Figure 6: TQP's represents data in tensors

```
statement = \
    "SELECT Digits, Sizes, COUNT(*)
    FROM numbers GROUP BY Digits, Sizes"

compiled_query = \
    tqp.sql.query.spark(statement, device="cuda")

compiled_query.run(toPandas=True)
```

Following is an overview of the supported operators in TQP:

(1) Relational operators such as selection, projection, sort, group-by aggregation (sort-based), natural join (hash and sort-based), non-equi, left-out, left-semi and left-anti joins.
(2) Comparison and arithmetic operators, date functions and Nulls.
(3) Statements such as `IN`, `CASE` and `LIKE` clauses.
(4) Aggregates like `SUM, AVG, MIN, MAX, COUNT, DISTINCT COUNT`.
(5) Scalar, Nested and Correlated subqueries.
(6) Prediction Queries: TQP provides seamless integration with PyTorch models and traditional ML models using Hummingbird [7], enabling native support for predictive capabilities. A Prediction Query, which encapsulates a trained ML model making predictions on the input data, can feature a combination of ML operations such as tree ensemble, one-hot encoding, scaling, and concatenation. These operations may be complemented by relational operators like join, aggregation, and filtering to create a comprehensive predictive model.

5 Evaluation

5.1 Quantitative Comparison

We benchmarked these databases on both the SSB and TPCH benchmarks and compared the performance of these databases with each other as well as DuckDB, which is a common OLAP CPU-based database. The performance of these databases on different SSB and TPCH queries can be found in Figure 7. For the TPCH benchmark, we were originally planning to benchmark it using SF = 8 but we could only find data for TQP, which is closed source, for SF = 1.

Looking at the performance on the SSB (SF = 16) queries, we see that DuckDB overall takes the most time followed by BlazingSQL and TQP taking the same amount of time, HeavyDB generally performs a little better than them with Crystal+ outperforming all of these systems substantially. Looking at Q11, Q12, and Q13 queries we see that BlazingSQL takes around 18.5 ms while TQP takes around 8.5 ms even though they perform equally on the other queries. For Q21, Q22, and Q23 we see all of the GPU databases have equal runtimes with limited variances but DuckDB runtimes range from 150 to 250 ms. For Q31, Q32, Q33, and Q34 we see that Crystal+ takes only 1.1 ms for Q32, takes around 1.9 ms for Q33 and Q34 and takes 2.5 ms for Q31.

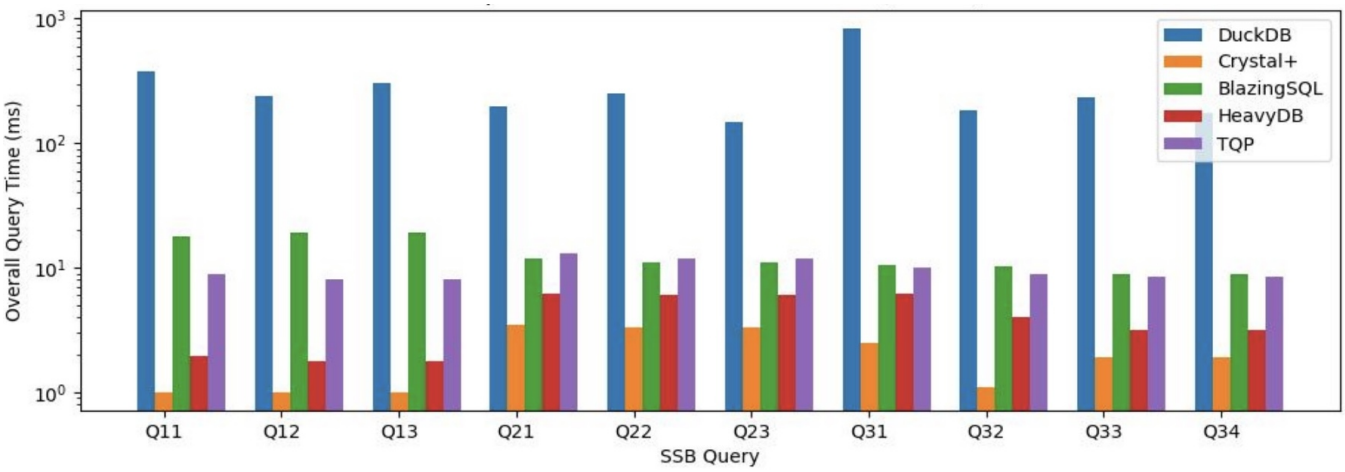
Looking at the TPCH (SF = 1) queries, there are a couple of things that stand out. Note that we used SF = 1 because of the fact that TQP only reported numbers for SF = 1 and since it is a closed source we can't benchmark its performance on other queries. We see that for Q4, DuckDB takes 216 ms while HeavyDB takes 292 ms and for Q13 DuckDB takes 181 ms while BlazingSQL takes 303 ms. This is especially visible in Q16 as DuckDB takes 93 ms, HeavyDB takes 3689 ms and TQP takes 301 ms. This is interesting that some of the leading GPU databases actually take more time than DuckDB, a common CPU-based OLAP solution for complicated queries as TPCH queries are generally considered more challenging than SSB.

One thing that is common across both of these benchmarks is that Crystal+ outperforms all of these databases. We see on the SSB benchmark that HeavyDB takes on average 4.052 ms while Crystal on average takes 2.05 ms which is a 1.97x improvement. This difference is especially pronounced in the TPCH benchmark where TQP takes 66.22 ms while Crystal+ takes 3.75 ms, which is a 17.66x improvement.

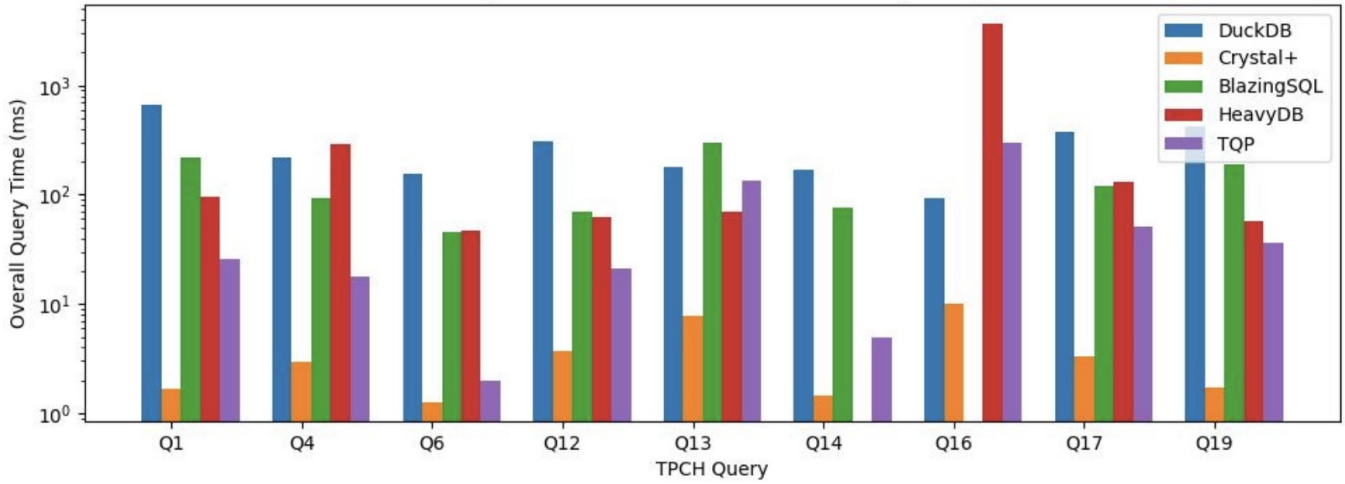
5.2 Qualitative Comparison

A qualitative comparison of these systems can be found in Table 1. The comparative analysis reveals distinctive characteristics among the considered database systems. BlazingSQL, deeply integrated with the RAPIDS ecosystem and cuDF, optimizes query compilation through GPU acceleration and in-memory operations. It particularly excels in large-scale data analytics and machine learning, emphasizing interoperability with RAPIDS libraries. HeavyDB, or OmniSciDB, stands out for analytical platforms and geospatial data support, leveraging GPU acceleration for high performance and featuring server-side rendering. CrystalDB targets SaaS providers with its multi-tenancy management, security, and compliance features, making it suitable for multi-cloud deployments. TQP uniquely focuses on analytical DBMSs and AI workloads, employing SQL-to-tensor program transformations and machine-level code compilation for efficient tensor computation.

In terms of data processing, BlazingSQL utilizes `GPU DataFrames`, while HeavyDB emphasizes vectorization and advanced memory management. CrystalDB boasts auto-scaling and optimization for speed and reliability, while TQP integrates SQL-to-tensor program transformations and machine learning. The performance of these systems is context-dependent, with BlazingSQL showcasing optimized query compilation and HeavyDB featuring rapid query compilation and native SQL support. CrystalDB excels in managed configuration for performance optimization, while TQP demonstrates high performance comparable to GPU systems over CPU. Each system brings special features to the table, such as BlazingSQL's interoperability with RAPIDS libraries and external data source integration, HeavyDB's server-side rendering and licensed web-based



(a) Comparison on the SSB benchmark (SF = 16)



(b) Comparison on the TPCH benchmark (SF = 1)

Figure 7: Overall runtime of various databases on many benchmark queries

visualization, CrystalDB's emphasis on multi-tenancy and security, and TQP's support for relational operations and machine-level code compilation. The development effort varies, with BlazingSQL prioritizing SQL compatibility and ecosystem integration, HeavyDB requiring a license for its full feature set, CrystalDB being designed for self-management with optional DBA involvement, and TQP featuring parsimonious engineering effort with portability across hardware. These distinctions underline the nuanced suitability of each system for specific use cases and deployment scenarios.

Note that we only compare complete GPU systems and thus don't include Crystal because it is a GPU library rather than a complete GPU database with features such as query plan. Additionally, these comparisons are based on the opinions of the author and are more subjective.

6 Conclusion

This paper provides an in-depth overview of four distinct systems designed to enhance database performance through GPU acceleration: BlazingSQL, OmniSciDB, Tensor Query Processor (TQP), and Crystal+. Each section explores the features, implementation details, and GPU utilization in executing queries for these systems, illustrated with examples. Performance comparisons on TPCH and SSB benchmarks, along with a qualitative assessment, are included.

To conduct a more comprehensive quantitative analysis, open-sourcing all systems is imperative, although currently unavailable. The goal is to delve into metrics like GPU/CPU utilization, IO operations, and performance across diverse benchmarks and sizes. Further exploration encompasses aspects such as reliability, transaction guarantees, security, privacy, scalability and simulating production-level workloads to evaluate the associated costs. As GPU databases are in their early stages, ongoing exploration aims to understand

Feature/Aspect	BlazingSQL	HeavyDB (OmniSciDB)	CrystalDB	Tensor Query Processor (TQP)
Underlying Tech	RAPIDS Ecosystem, Apache Calcite, cuDF	GPU acceleration, SQL engine, geospatial support	Serverless PostgreSQL	Tensor Computation Runtimes (e.g., PyTorch, TVM)
Data Processing	Using GPU DataFrames	High performance, vectorization, advanced memory mgmt	Auto-scaling, optimized for speed and reliability	Columnar Tensor-based data format
Performance	Optimized Query Compilation, In-Memory operations and GPU acceleration	Rapid query compilation, native SQL support	Managed configuration for performance optimization	High performance over CPU, comparable to GPU systems
Special Features	Interoperability with RAPIDS libraries and External Data Source Integration	Server-side rendering, web-based visualization (licensed)	Multi-tenancy management, security, compliance	Unified runtime for relational and ML operators; hardware agnostic
Use Case	Large-Scale Data Analytics and Machine Learning with cuML Integration	Analytical platforms, geospatial data	SaaS providers, multi-cloud deployments	ML assisted Analytical workloads
Development Effort	SQL Compatibility and Integration with Existing Ecosystems	Requires a license for full feature set	DBA optional, self-managing	Minimal engineering effort, Closed-source and in development phase.

Table 1: Comparative Architectural Features

limitations hindering production deployment and strategize solutions to overcome these challenges.

References

[1] [n.d.]. Dictionary encoding - AnalyticDB for PostgreSQL - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/analyticdb-for-postgresql/user-guide/dictionary-encoding

[2] BlazingDB. 2023. *BlazingSQL is a lightweight, GPU accelerated, SQL engine for Python.* https://github.com/BlazingDB/blazingsql

[3] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. Revisiting Query Performance in GPU Database Systems. arXiv:2302.00734 [cs.DB]

[4] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesú s Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proceedings of the VLDB Endowment* 15, 11 (jul 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[5] Heavy.AI. 2023. *Open Source Analytical Database SQL Engine.* https://www.heavy.ai/product/heavydb

[6] HeteroDB. 2023. *PG-Strom is an extension for PostgreSQL database.* https://github.com/heterodb/pg-strom

[7] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 899–917. https://www.usenix.org/conference/osdi20/presentation/nakandala

[8] NVIDIA. 2023. Cub. https://github.com/NVIDIA/cub.

[9] NVIDIA. 2023. cuCollections. https://github.com/NVIDIA/cuCollections/tree/dev.

[10] NVIDIA. 2023. Thrust. https://github.com/NVIDIA/thrust.

[11] P E O'Neil, E J O'Neil, and X Chen. 2009. The Star Schema Benchmark (SSB).

[12] Meikel Poess. 2019. TPC-H. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. http://dblp.uni-trier.de/db/reference/bdt/1.html#Poess19a

[13] PostgresSQL. 2023. Explain. https://www.postgresql.org/docs/current/sql-explain.html.

[14] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (dec 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[15] 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* (Portland, OR, USA) *(SIGMOD '20).* 1617–1632. https://doi.org/10.1145/3318464.3380595

[16] 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.

[17] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-Based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22).* Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132