Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems

MARKO KABIĆ, Department of Computer Science, ETH Zurich, Switzerland
SHRIRAM CHANDRAN, Department of Computer Science, ETH Zurich, Switzerland
GUSTAVO ALONSO, Department of Computer Science, ETH Zurich, Switzerland

Several trends are changing the underlying fabric for data processing in fundamental ways. On the hardware side, machines are becoming heterogeneous with smart NICs, TPUs, DPUs, etc., but specially with GPUs taking a more dominant role. On the software side, the diversity in workloads, data sources, and data formats has given rise to the notion of composable data processing where the data is processed across a variety of engines and platforms. Finally, on the infrastructure side, different storage types, disaggregated storage, disaggregated memory, networking, and interconnects are all rapidly evolving, which demands a degree of customization to optimize data movement well beyond established techniques. To tackle these challenges, in this paper, we present Maximus, a modular data processing engine that embraces heterogeneity from the ground up. Maximus can run queries on CPUs and GPUs, can split execution between CPUs and GPUs, import and export data in a variety of formats, interact with a wide range of query engines through Substrait, and efficiently manage the execution of complex data processing pipelines. Through the concept of operator-level integration, Maximus can use operators from third-party engines and achieve even better performance with these operators than when they are used with their native engines. The current version of Maximus supports all TPC-H queries on both the GPU and the CPU and optimizes the data movement and kernel execution between them, enabling the overlap of communication and computation to achieve performance comparable to that of the best systems available, but with a far higher degree of completeness and flexibility.

CCS Concepts: • **Information systems** → **DBMS engine architectures**; **Relational parallel and distributed DBMSs**; **Query operators**; **Main memory engines**; *Relational database model*; Online analytical processing engines; Record and buffer management; **Database query processing**; • **Computer systems organization** → **Data flow architectures**; **Heterogeneous (hybrid) systems**; • **Software and its engineering** → *Abstraction, modeling and modularity*.

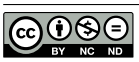
Additional Key Words and Phrases: Database Systems, Composable Query Engines, Modular Query Engines, OLAP, Vectorized Execution, Parallel Execution, GPU acceleration, Heterogeneous Systems

ACM Reference Format:
Marko Kabić, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 187 (June 2025), 25 pages. https://doi.org/10.1145/3725324

1 Introduction

Several transformative trends are fundamentally reshaping the landscape of data processing. On the hardware front, there is a shift toward increasingly heterogeneous systems. Modern machines now commonly incorporate Network Interface Cards (NICs) and a variety of accelerators such as Graphic Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Tensor Processing

Authors' Contact Information: Marko Kabić, marko.kabic@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Shriram Chandran, schandran@student.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Gustavo Alonso, alonso@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland.



ACM 2836-6573/2025/6-ART187
https://doi.org/10.1145/3725324

Units (TPUs), and smart NICs, as well as other specialized hardware at the processor level (AVX units, encryption engines and compression cores, etc.). GPUs have taken a particularly dominant role due to their exceptional parallel processing capabilities and high memory bandwidth. Additionally, the growing computational demands of AI and ML workloads have led to the widespread deployment of GPUs in data centers, opening the opportunity to use them for other workloads.

On the software side, workloads are becoming more varied, encompassing everything from traditional transactional and analytical workloads, to graph analytics [32, 59], to complex machine learning tasks such as data pre-processing and ingestion [60], data cleaning [18], training and inference [35, 49, 54]. This heterogeneity of hardware, coupled with the workloads diversity has led to the development of highly optimized, yet compartmentalized, data processing frameworks, each optimized for particular types of workloads and hardware. Although these systems have good performance, they often operate in isolation, limiting their ability to leverage the full spectrum of available hardware resources.

The complexity of this heterogeneity gave rise to the development of modular and composable Data Management Systems (DMS) and data processing frameworks, such as CVM [38], Modularis [29], LingoDB [24], Velox [43], Datafusion [30], to name a few. Although these frameworks were focused on modularity and composability, their design choices do not allow the existing implementations of data processing operators from third-party engines to be easily integrated and reused within these frameworks. In order to allow efficient use of existing operator implementations from different engines, Apache Gluten [17, 52] has been developed, which splits a query plan into query plan fragments and offloads them to native engines. Although planned, the GPU backend is not yet available in Gluten. Similarly, BOSS [37] enables the query execution to be split among multiple engines, by using the concept of partial query evaluation. While this method enables the use of various engines, it introduces some overhead and, more importantly, does not allow efficiently overlapping communication and computation between different devices, which is especially important in heterogeneous systems, where data transfer becomes a bottleneck [11, 12] (see Section 2).

To address these challenges, we have developed Maximus: a GPU-accelerated query engine written purely in C++, enabling the execution of arbitrary hybrid CPU-GPU query plans, allowing efficient overlap of communication and computation, especially between the CPU and the GPU. To enable the use of existing operators from third-party engines, we introduce the concept of *operator-level integration*, which allows third-party operators to be used as native operators. To demonstrate this concept in Maximus, we integrate the operators from Apache Arrow Acero (CPU) and RAPIDS cuDF (GPU), together with Maximus' own native operators. This operator-level integration of third-party engines is achieved through a modular and composable architecture that allows operators from different engines to be seamlessly combined and swapped on an operator level, enabling the utilization of the extensive optimizations and specialized functionalities of existing engines. This low-level integration of third-party operators enables Maximus to take full control of the execution and further configure the parameters like the number of threads, the processing chunk sizes, the amount of pinned memory used and other parameters, which might not be fine-tuned or exposed by third-party engines via external APIs. In this way, Maximus can outperform third-party engines, even when it uses the same operators and the same hardware as these third-party engines, as demonstrated in the result section. In addition, the advantages of operator-level integration are further amplified in heterogeneous systems that include both CPUs and GPUs. These systems present additional opportunities for CPU-GPU co-processing as well as overlapping CPU-GPU data transfers with query execution. In this work, we propose strategies for achieving these capabilities.

Designing such a unified framework, however, brings many challenges. First, different engines use different in-memory data structures during processing. For example, Acero uses an `acero::`

`ExecBatch` data structure whereas cuDF uses `cudf::table`. In addition to table data structures, each of these engines has its own set of data types and expressions, often not directly compatible. Second, the engines employ different execution strategies, e.g. Acero supports streaming execution, which is lacking in dataframe libraries like cuDF. Third, each of these engines has its own operator interface. Finally, each of these engines uses its own memory pool and its own execution environment, containing the thread pool, GPU stream pools and other execution-related objects.

Building a truly composable query processing engine requires addressing all these challenges in a flexible and extensible manner without sacrificing performance. In Maximus, we address these challenges by making the following contributions:

(1) We develop the concept of *operator-level integration*, which enables operators from multiple engines to be used efficiently within the same executor, as native operators.
(2) We extend the idea of morsel-driven parallelism to heterogeneous systems and propose strategies for efficiently overlapping communication and computation at multiple levels.
(3) We define a unified in-memory data representation, to capture the table representations from different CPU and GPU engines. Moreover, our table representation offers 0-copy, asynchronous, and lazy conversions to other formats, whenever applicable.
(4) We define a unified operator interface that supports both streaming and bulk execution. This allows us to integrate both streaming engines like Apache Arrow Acero, as well as dataframe libraries like RAPIDS cuDF.
(5) We introduce the idea of a proxy operator layer isolating third-party engine operators so that they can be used as native Maximus operators. This enables third-party operators to be treated as kernels, rather than operators, and ensures that all operators use the same memory pool, whenever applicable.
(6) We extend the modular architecture design studied before [29, 30, 37, 38, 43, 44], to support hybrid CPU and GPU execution, capable of utilizing the CPU-GPU parallelism and overlap the communication with computation more efficiently.
(7) We evaluate Maximus performance, showing that it enables multiple engines to complement each other by fully utilizing the available hardware with a modular and composable design.

Maximus architecture facilitates the adoption of new hardware capabilities without necessitating extensive rewrites of existing codebases. By open-sourcing the Maximus codebase, we invite the community to integrate operators from other systems like DuckDB [46] or Velox [43].

2 Related Work

Data processing on modern hardware has been attracting a lot of interest lately, including data analytics on GPUs [3, 6, 8, 50, 58], FPGAs [13, 23, 28, 33, 36, 51], and even TPUs [21]. The evolving hardware developments coupled with the increasing complexity of Data Management Systems (DMS) have led to a growing interest in composable and modular DMS architectures in recent years [4, 29, 30, 38, 39, 44], proposing different methods to achieve modularity and composability. The idea of defining the operators at a finer granularity, the so-called (sub)operators can be found e.g. in the works of Bandle and Giceva [4], Kohn et al [26], Dittrich and Nix [14]. Some of these ideas were further extended and adopted by frameworks supporting compiled execution like CVM [38], Modularis [29] and LingoDB [24]. Although these approaches have demonstrated that defining the operators at a finer granularity can indeed allow additional flexibility in terms of supporting diverse workloads (e.g. linear algebra and OLAP) [38], different distributed platforms (RDMA clusters and Serverless infrastructures), and smart storage solutions (e.g. smart storage) [29], none of these supports the use of accelerators like GPUs in practice. The main challenge here lies in the fact

that CPUs and GPUs operate at different granularity, making it hard to define a unified set of sub-operators that would work for both types of hardware.

A similar trend of designing modular and composable frameworks can be observed among the vectorized engines [44]. Apache Datafusion [30] defines a set of extension APIs, allowing customization of the infrastructure on multiple levels, although, without supporting GPUs at the time of writing. The Meta's Velox engine [34, 43] aims to provide an execution engine capable of processing diverse workloads in a unified way.

Although these frameworks offer an extensible mechanism to enhance their functionality, such as e.g. adding new operators, they cannot efficiently leverage existing operators from other engines, leading to a redundant effort of reimplementing the operators in each engine.

In order to allow using multiple engines during the query execution, Apache Gluten [17] was introduced. It splits a query plan into multiple fragments, offloading each of these fragments to different native engines. Although planned, the GPU support is not available at the time of writing. Similarly, BOSS [37] allows multiple engines to be used during the query execution through the concept of partial query evaluation, which they introduce. This is achieved by a query being passed through a sequence of stages, where in each stage, one of the integrated engines can partially evaluate the query by executing one or more operators. This partially evaluated query is then passed through other engines until it completes.

Although the approach of partial query evaluation allows the existing engines (e.g. a CPU engine and a GPU engine) to be integrated and combined with minimal implementation effort, it imposes certain limitations on the execution, which can affect the performance, particularly in hybrid CPU-GPU systems. Firstly, passing the query through a sequence of engines imposes an implicit sequential execution among different engines. Although multiple operators can be processed concurrently within the same engine at each stage, this sequential flow prevents the parallel execution of operators across different engines. This limitation can degrade performance in two common scenarios: (1) in wide query plans, numerous independent operators could ideally execute in parallel, potentially across different engines and hardware (e.g., CPUs and GPUs); (2) when scheduling multiple queries for execution, numerous independent operators should also be able to run in parallel on potentially different engines. Secondly, the implicit sequential execution of stages further prevents the overlap of communication and computation, which is especially important for optimizing performance in hybrid CPU-GPU systems. While numerous strategies have been proposed to efficiently leverage the CPU-GPU parallelism during query execution [9, 27, 48], none of these frameworks provide an effective mechanism for integrating and utilizing operators from third-party engines.

Our approach addresses these limitations by introducing the concept of *operator-level integration*, as opposed to the partial query evaluation, which enables the integration of operators from multiple engines, on a lower level. This allows our executor to achieve faster performance than the native executors of these operators, while preserving the modular and composable design.

3 Operator-Level Integration

The operator-level integration provides a mechanism to efficiently integrate operators from third-party engines, allowing them to be used together with the native operators by the Maximus executor. This requires interoperability across operators from different engines and on heterogeneous hardware, posing significant semantic and execution-related challenges.

Table 1. The in-memory data formats supported in Maximus.

Type	CPU Formats	GPU Formats

Batch-like	`arrow::RecordBatch`
	`acero::ExecBatch`
		`cudf::table`
		`cudf::table_view`
		`maximus::GTableBatch`

Table-like	`arrow::Table` `maximus::Table`	`maximus::GTable`

3.1 Interoperability Challenges

3.1.1 Semantic challenges. The semantic challenges include dealing with subtle semantic differences between data types, data representations, expressions, functions (e.g. scalar functions, user-defined functions (UDFs), aggregate functions and window functions), and relational operator interfaces and behavior. For example, when casting the floating-point number 1.7 to an integer, some engines may return 2, while others might return 1. Such inconsistencies are not limited to functions; they also extend to relational operators. For instance, a grouped aggregation operator might assume a non-empty key set in some systems, whereas others allow for an empty key set. To cope with these challenges and allow interoperability across different languages and systems, the *Substrait*[53] format has been developed with the aim to define the semantics of data types, expressions, functions and relational operators on both logical and physical levels. While not perfect and still evolving, it is a good basis to build heterogeneous systems as demonstrated by the wide adoption in major query engines, including DuckDB, Apache DataFusion, Apache Acero, Velox, and others. Maximus uses the Substrait semantics and defines Substrait-compatible types, expressions, functions and relational operators. This allows it to integrate operators from any Substrait-compatible engine, improving consistency. Similar semantic challenges arise in the industry efforts towards standardization and interoperability [1, 7], such as a recent trend towards unified lakehouse systems that integrate diverse engines and data formats, frequently relying on Substrait. This trend faces the same problems with diverse semantics like Maximus, indicating that the industry is now moving towards addressing them through convergence on formats like Substrait and common data representations such as Apache Iceberg [56], which we are incorporating into Maximus as part of ongoing and future work.

3.1.2 Execution-related challenges. These challenges include dealing with scheduling and resource management on systems with heterogeneous hardware, including managing the memory, thread pools, GPU streams, contexts, synchronization and similar. Maximus provides an execution framework that manages all these aspects of the query execution and resource management on heterogeneous systems, while keeping the semantic compatibility with Substrait. The coverage of operators in Maximus can help Substrait better understand the issues arising in practice.

3.2 Types, Expressions, Functions

Maximus defines Substrait-compatible data types, expressions and functions (including the aggregate functions) and provides conversions to Apache Arrow (CPU) and RAPIDS cuDF (GPU) types, expressions and functions. Providing such conversions is challenging since the set of supported aggregate functions and expressions varies among engines. For example, cuDF was missing expressions like `if_else`, that were necessary for supporting the full TPC-H. In such cases, we implemented the missing cuDF expressions using lower-level cuDF primitives.

```cpp
class DeviceTablePtr {
    using unified_table_t = std::variant<
        Empty,
        shared_ptr<arrow::Table>,
        shared_ptr<arrow::RecordBatch>,
        shared_ptr<acero::ExecBatch>,
        shared_ptr<maximus::TableBatch>,
        shared_ptr<maximus::Table>,
        shared_ptr<cudf::table_view>,
        shared_ptr<cudf::table>,
        shared_ptr<maximus::Gtable>,
        shared_ptr<maximus::GTableBatch>,
    >;
    unified_table_t original_table;
    unified_table_t table;

    template <typename T>
    void convert_to<T>();
    // other methods
};
```

Listing 1. DeviceTablePtr wraps all supported data formats within an `std::variant` and allows efficient conversions between any supported data formats.

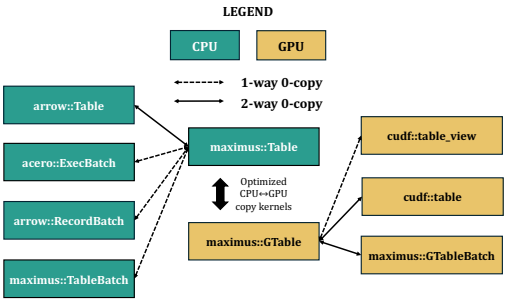


Fig. 1. Maximus offers all-to-all conversions for all supported data formats. These conversions are 0-copy whenever possible.

3.3 Unified Data Representation

Maximus supports a wide range of columnar, in-memory data formats (Table 1) enabling data exchange between operators from different engines. Supporting both batch-like and table-like data formats, which allows both streaming and blocking operators to be integrated:

- **`arrow::Table`:** represents the de-facto standard for columnar, in-memory, tabular data layout. Today, many state-of-the-art engines support 0-copy conversion between the `arrow::Table` and their internal (native) data formats. It can store very large columns, by breaking them into smaller chunks of consecutive elements.
- **`arrow::RecordBatch`:** similar to `arrow::Table`, but here each column consists of only a single chunk of consecutive elements.
- **`maximus::Table`, `maximus::TableBatch`:** these are thin wrappers around `arrow::Table` and `arrow::RecordBatch`, mostly aimed at being used by native Maximus operators.
- **`acero::ExecBatch`:** the data format used internally by Apache Acero similar to `arrow::RecordBatch`, but with additional optimizations, such as run-length encoding. For example, columns consisting of a single constant literals are stored as a single value instead. Moreover, it can contain a selection map, which marks the values that should be preserved, which can be used e.g. to optimize the row filtering.
- **`cudf::table`:** a columnar data layout on the GPUs, similar to `arrow::RecordBatch`, used by the RAPIDS cuDF library. Although it is called a table, it can only contain a single batch.
- **`cudf::table_view`:** a read-only, non-owning version of a `cudf::table`. Most cuDF operators take a `cudf::table_view` as the input and produce a `cudf::table` as the output.
- **`maximus::GTable`, `maximus::GTableBatch`:** data format similar to `cudf::table`, but with the CUDA-dependent parts abstracted away, with the aim of supporting AMD GPUs at a later point.
- **DeviceTablePtr.** All these types are wrapped inside a `DeviceTablePtr`, which is an `std::variant` of all these data formats, as shown in Listing 1. All data is passed between operators as `DeviceTablePtr` objects, which allows the operators to internally choose which data format to use.

```
class AbstractOperator {
  AbstractOperator(
    shared_ptr<Context> ctx,    // global context
    vector<Schema> in_schemas, // input schemas
    Properties properties);    // substrait-compatible properties

  // the kernel methods
  void add_input(DeviceTablePtr input, int in_port);

  void no_more_input(int in_port);

  bool has_more_batches(bool blocking);

  DeviceTablePtr export_next_batch();

  // other methods
};
```

Listing 2. An operator interface as used by Maximus.

When converting between different data formats in Maximus, the conversion is zero-copy, asynchronous and lazy whenever possible:

- **Zero-copy.** `DeviceTablePtr` also comes with an efficient function that can convert between any supported data formats. The conversion between the supported formats is 0-copy, whenever possible, as shown in Figure 1.
- **Asynchronous copy.** Conversions between CPU and GPU data formats involve CPU↔GPU data transfers. All CPU↔GPU data transfers are asynchronous, to allow overlapping communication and computation, whenever possible. Due to these asynchronous data transfers, the `DeviceTablePtr` has to maintain the pointer to the original table (`original_table`), until the data transfer is complete and the table variable (`table`) is ready to be used.
- **Lazy conversion.** All conversions between different table formats on the same hardware type (CPU or GPU) are lazy. This prevents unnecessary data format conversions between consecutive operators. For example, consecutive Acero operators will keep the data as `acero::ExecBatch`, wrapped up within a `DeviceTablePtr`, without the overhead of converting them to and from a `maximus::TableBatch` when passing the data between each other.

3.4 Unified Operator Interface

3.4.1 Operator Interface. For each operator, Substrait defines the operator's arguments by defining *properties* that include all the parameters necessary to define the operator's semantics. For example, for a Hash-Join physical operator, Substrait properties include the left and the right keys, an optional post-join predicate as well as the join type.

Maximus defines a simple `AbstractOperator` interface as shown in Listing 2. The constructor includes the `Properties` object that is substrait-compatible. All the execution-related objects, such as e.g. a memory pool, a thread pool, GPU streams etc. are grouped within the global context object that is shared among all the operators and alive throughout the query execution.

```
class FusedOperator : public AbstractOperator {
  FusedOperator(
    shared_ptr<Context> ctx,         // global context
    vector<Schema> in_schemas,       // input schemas
    vector<Properties> properties); // a sequence of properties
};
```

Listing 3. A `FusedOperator` interface that semantically represents a sequence of operators.

3.4.2 Operator Functionality. In most engines, the methods which describe the operator functionality can be split into 3 categories:

(1) the executor-independent methods for adding the input data and retrieving the output data;
(2) the executor-dependent methods for controlling the execution;
(3) the DAG information containing pointers to the neighboring operators (e.g. upstream and/or downstream operators).

These categories are visualized in Figure 2. This architecture makes the operators dependent on their native executors. In Maximus, we propose a more modular design that separates these methods into different abstractions: kernels, operators and query plan nodes. This design allows Maximus to use third-party operators within the Maximus executor through a proxy operator layer, described below.

3.4.3 Kernels. A kernel logically resembles a mathematical function that receives some input and computes the output. In Maximus, the kernel methods include the methods for adding the input (`add_input`), signaling that all the input has been received (`no_more_input`) and the methods for extracting the kernel output: `has_more_batches` for checking if there are any output batches being produced and `export_next_batch` for retrieving the operator's output. Retrieving the operator's output by invoking the `export_next_batch` transfers the ownership of the output batch back to the caller (i.e. the executor).

3.4.4 Operators. An operator in Maximus contains the kernel methods, with other functionalities that are relevant to the executor. This includes the information about the device (CPU or GPU) on which the operator executes, the engine type (e.g. Acero, cuDF or native), and the global context containing the memory pool and other execution-dependent resources. The operator however, does not contain any information about other operators in a query plan.

3.4.5 Fused Operators. Some engines often combine multiple operators into a single operator. For example, a filter and a project operators are often combined into a single FilterProject operator, as found e.g. in Velox. Similarly, a hash-join can sometimes include a post-join predicate (i.e. a filter), as found in Apache Arrow Acero.

To allow the integration of such operators, Maximus defines a `FusedOperator` class, which has a similar interface as the `AbstractOperator` (see Listings 2 and 3), with the difference that it takes a sequence of `Properties`, as it semantically represents a sequence of operators, as shown in Listings 3.

3.4.6 Query Plan Nodes. A node contains the operator, as well as the pointers to all the downstream and upstream nodes in a query plan. This allows the query planner to rewrite or restructure the execution DAG without affecting the operators. Maximus adds a special (empty) query node at the

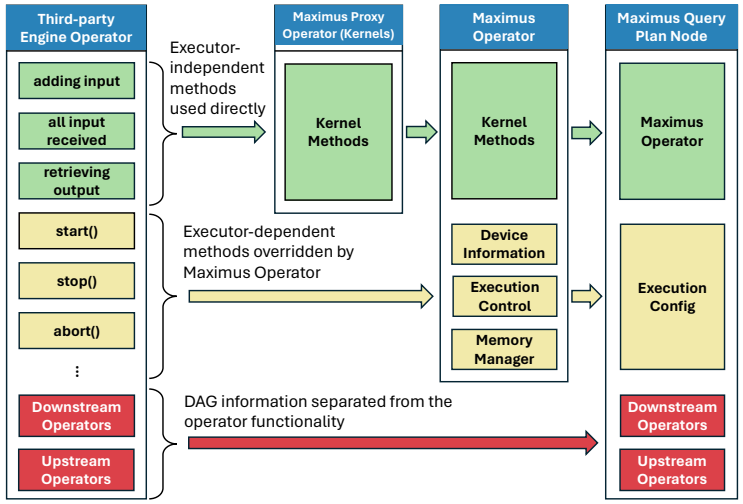


Fig. 2. Maximus modular design splits typical operator methods into different abstractions, allowing it to use third-party engine operators within the Maximus-native executor.

root of the execution DAG that is called *QueryPlan*. This `QueryPlan` object initiates the schema inference and query rewritings that are then propagated throughout the whole excution DAG.

3.4.7 Proxy Operator Layer. In order to use third-party operators within the Maximus executor, they need to be used as kernels, rather than operators. In Maximus, this is achieved by adding a proxy operator layer, that only uses the kernel methods from a third-party engine operator and creates an isolated execution environment and, if needed, dummy downstream and upstream operators. The proxy operator layer also provides a proxy memory pool that intercepts all the memory allocations and deallocations from third-part operators, enforcing all the operators to use the Maximus memory pool, which we describe below.

3.4.8 Proxy Memory Pool. Maximus internally uses the arrow's memory pool for all CPU allocations and the RAPIDS' memory pool for all GPU allocations. If a third-party operator uses a different memory pool, we use the proxy design pattern to intercept all the memory allocations and deallocations with the memory pools used in Maximus. For example, if third-party operators use `ThirdPartyMemoryPool`, we make them use the Maximus memory pool by creating a `ProxyMemoryPool` that inherits the `ThirdPartyMemoryPool` and overrides all the methods. This `ProxyMemoryPool` is then passed to all the operators using this pool, to ensure all the allocations are using the default Maximus memory pools.

3.5 Unified DataFlow Interface

Physical query plans are usually represented as a DAG (or a tree) of physical operators. Some engines, like DuckDB [46, 47] and Acero [15] allow the physical operators like hash join, to have multiple inputs or ports, where each input can be streaming or blocking. We refer to this design as *multi-port operator design*.

Other engines, like Velox [43] and Modularis [29], split the multi-port operators into sub-operators during the execution, where each sub-operator usually has only 1 input and 1 output edge. For a hash join operator, it is split into a build operator, which is blocking, and a probe operator, which is streaming. We refer to this design as the *sub-operator design*.

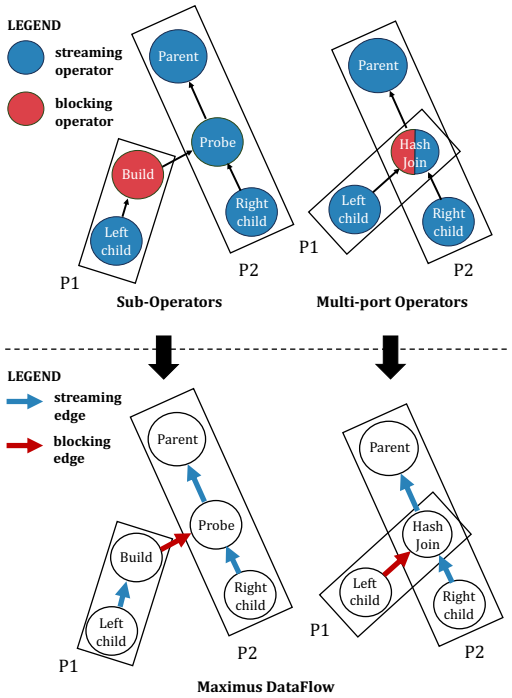


Fig. 3. Maximus proposes a more-general dataflow representation, allowing it to integrate sub-operators as well as the multi-port operators from third-party engines.

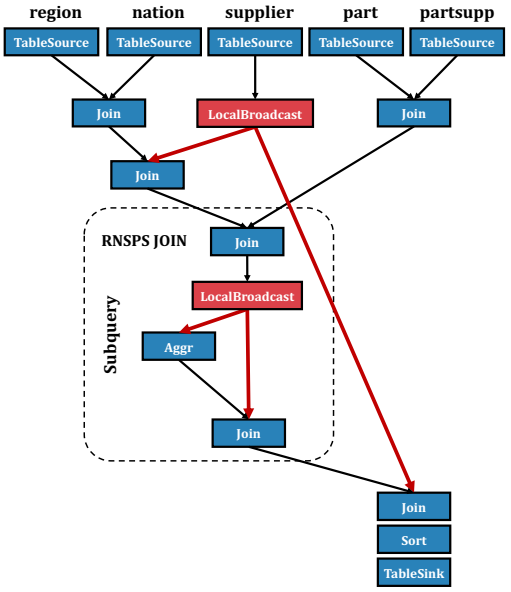


Fig. 4. Maximus query plan for the Q2 TPC-H query. DAG-shaped query plans can be executed by using a special *LocalBroadcastOperator* which can have multiple outgoing edges.

In order to allow the integration of both types of physical operators in Maximus, we propose a unified dataflow representation (see Figure 3), where each physical operator can have multiple input ports and each edge (instead of an operator) can be streaming or blocking. The Maximus executor then ensures that all the input from the blocking edges is received before the input from the streaming edges starts propagating. In this way, both types of operators from other engines can easily be uniformly integrated.

3.5.1 Non-tree Plans. Many query engines make an implicit assumption that a query plan is a tree, thus restricting each node in a physical query plan to have at most one outgoing edge. However, in practice, DAG-shaped query plans, which are not trees, can commonly occur, especially when subqueries are involved. For example, Q2, from the TPC-H [57] benchmark, contains multiple operators with 2 outgoing edges, as shown in Figure 4. In Maximus, any DAG-shaped query plans can be executed by introducing a special *LocalBroadcastOperator*, which accepts incoming batches and replicates them to multiple output ports. It is available for both the CPU and the GPU, allowing the execution of arbitrary DAG-shaped query plans on both types of hardware.

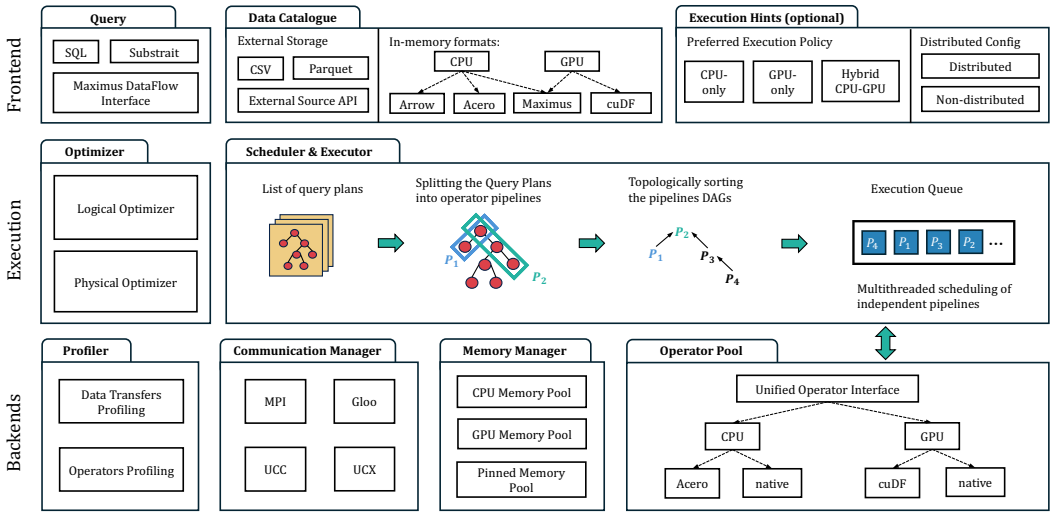


Fig. 5. The main components of the Maximus query engine architecture (Section 4), including the frontend layer (top), the execution layer (middle) and the backends layer (bottom).

4 Maximus Architecture

4.1 Design Overview

Maximus is a GPU-accelerated query engine, written purely in C++ that is available as an open-source library[1]. Its architecture, depicted in Figure 5, is composed of multiple layers:

(1) **Frontends:** contains the components for specifying the queries, data sources and optionally, execution hints.
(2) **Execution Layer:** contains the main components controlling the execution, which includes the optimizer, the scheduler and the executor.
(3) **Backends:** contains a variety of available implementations of operators (Operator Pool), communication routines (Communication Manager) as well as the Memory Manager, all of which are directly used by the execution layer.

The architecture in Figure 5 contains some elements of the distributed version of Maximus, which is beyond the scope of this work. Below, we discuss the most important components in more detail.

4.2 Frontends

The Frontends Layer contains components for defining the query, the data sources and optionally the execution hints, allowing the user to have full control of the execution.

4.2.1 Query. A query can be defined in SQL, Substrait format [53] or using the native Maximus DataFlow Interface. The SQL is supported through the Hyrise SQL parser [19], that is integrated into Maximus. The Substrait format is supported through Acero, with the following conversion path: Substrait Plan → Acero ExecPlan → Maximus DataFlow Interface.

4.2.2 Data Catalogue. The data catalogue stores the information about the tables metadata. The tables can either reside in a file system (e.g. as CSV or Parquet files) or in the main memory in some

[1]https://gitlab.inf.ethz.ch/PUB-SYSTEMS/eth-dataprocessing/Maximus

of the supported in-memory CPU/GPU table data structures (see Section 3.3). The tables can be loaded from a file system either into the CPU memory, using the Arrow readers or directly into the GPU memory, using the cuDF readers.

4.2.3 Execution Hints. The user can, optionally, specify some additional execution hints. For example, the user can choose a preferred execution policy: CPU-only, GPU-only or a hybrid CPU-GPU. Maximus can execute arbitrary hybrid CPU-GPU execution plans. By default, the hybrid execution policy performs the initial filter and projection operators on CPUs and offloads the rest to the GPUs. In addition, the user can specify how many outer (scheduler) threads and how many inner (kernel) threads to use for the execution. These can be specified either through the global context object or through environment variables.

4.3 Optimizer

A query plan is a doubly-linked DAG, which can be easily traversed. This design allows adding arbitrary rewrite rules for query optimization. Moreover, the query node class contains the method `convert_to_physical()` which allows converting the operators to lower-level abstractions that might be closer to the hardware, as proposed in [25]. This design allows implementing a wide range of logical and physical optimizations.

4.4 Scheduler

The scheduler allows multiple query plans to be scheduled before they are executed. During scheduling, each query plan is split into pipelines where each pipeline is a sequence of operators that starts and ends with a pipeline breaker or a leaf or a root, in a similar way as described in the morsel-driven parallelism [31]. The scheduler ensures the pipelines that have blocking outgoing edges are executed before the pipelines that have streaming outgoing edges. Based on these constraints, a new DAG of pipelines (instead of single operators) is created. The pipelines in this DAG are then topologically sorted within the execution queue. Optionally, a task-based framework, such as Taskflow [22] can be used for scheduling the DAG of pipelines.

4.5 Executor

The executor executes the pipelines from the execution queue. It contains two levels of threads: (1) the outer threads (the scheduler threads) are used to execute independent pipelines in parallel; (2) the inner threads (the kernel threads) that are internally used by operators within each pipeline. The total number of threads is thus the product of the outer and the inner threads. The number of outer and inner threads can, optionally, be set through execution hints.

4.6 Operator Pool

The operator pool contains all the available operators. For each operator, there can be multiple available versions. For example, for a `FilterOperator`, there are two available versions: one from the Acero engine and one from the cuDF engine. For some other operators, like `LocalBroadcast Operator` there are two versions: the native CPU and the native GPU version.

All the available operator versions are grouped in namespaces, that are of the form: `<device>::<engine>::<operator>`. For example, the two available implementations of the filter operator are `cpu::acero::FilterOperator` and `gpu::cudf::FilterOperator`. Similarly, the two available implementations of the local broadcast operator are: `cpu::native::LocalBroadcastOperator` and `gpu::native::LocalBroadcastOperator`.

Table 2. A sample profiler output in Maximus.

Regions	Min time	Max time	...
OPERATORS	0.000900	0.000905	...
CPU::ACERO::HASH_JOIN	0.000898	0.000898	...
add_input	0.000497	0.000497	...
no_more_input	0.000372	0.000372	...
has_more_batches	0.000003	0.000003	...
export_next_batch	0.000002	0.000002	...
...	...	...	...
DATA MOVEMENTS	0.000082	0.000085	...
CPU->CPU	0.000037	0.000037	...
ArrowTable->AceroBatch	0.000015	0.000015	...
CPU->GPU	0.000042	0.000045	...
AceroBatch->CudfTable	0.000040	0.000040	...
...	...	...	...

This nomenclature allows a highly modular and composable design, where a concrete version of each operator can be easily switched just by changing the namespaces of the operators in a query plan. The Executor ensures the data transfers and data conversions are performed where needed.

4.7 Memory Manager

Maximus contains three important memory pools: a CPU memory pool using the arrow memory pool, a GPU memory pool using the RAPIDS cuDF's memory pool (`rmm`), and a pinned memory pool of the page-locked host memory. The proxy layer (see Section 3.4) ensures all the operators are using these memory pools.

4.7.1 Pinned Memory Pool. All CPU↔GPU data transfers are going through the pinned memory, to optimize the data transfers. This allows the Direct Memory Access (DMA) to directly copy the data, rather than having to go through intermediary pinned memory buffers.

4.8 Profiler

Maximus is integrated with the Caliper Performance Analysis Toolbox [5], enabling detailed profiling analysis with low overhead. The code annotations in Maximus clearly separate the time spent in executing the operators (computation), as well as the time spent in data transformations and all data movements between different devices (communication). When Maximus is compiled with the profiling flag, a detailed profiling output is produced, as shown in Table 2. In addition, an NVTX-compatible traces are produced, enabling them to be visualized with the standard NVIDIA Nsight Systems profiler [40].

5 Unlocked Performance Advantages

The concept of operator-level integration, together with the Maximus architecture, enables Maximus to take full control of the execution. This further allows implementing various mechanisms, aimed at efficiently leveraging CPU-GPU parallelism and overlapping query execution with CPU-GPU data transfers. Implementing these mechanisms in Maximus allows them to be seamlessly applied to both integrated third-party operators as well as native Maximus operators. In this section, we introduce two such mechanisms: *heterogeneous morsel-driven parallelism* and the *communication* (CPU-GPU data transfers) and *computation* (operator execution) *overlap*.

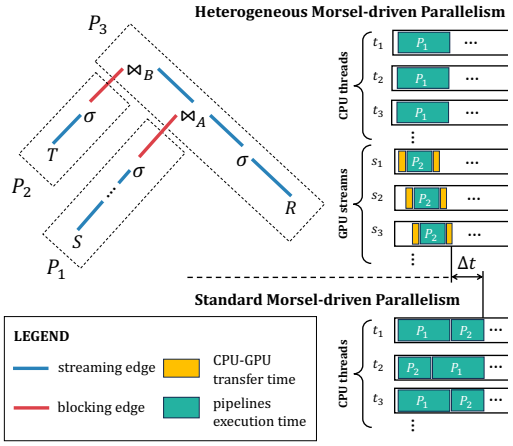


Fig. 6. In heterogeneous morsel driven parallelism, the data parallelism over pipelines can be applied to GPU streams, in addition to CPU threads.

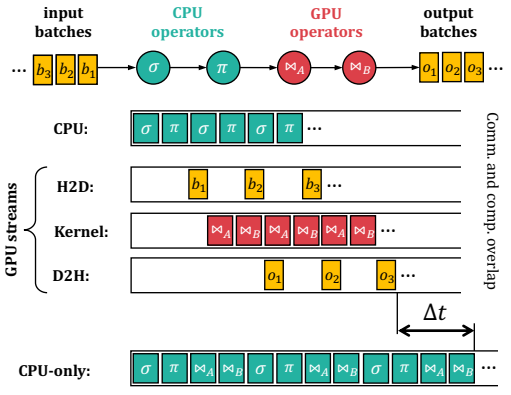


Fig. 7. The overlap of communication and computation between a CPU and a GPU.

5.1 Heterogeneous Morsel-Driven Parallelism

In the traditional morsel-driven parallelism model [31], a query is divided into pipelines, with multiple threads processing these pipelines using distinct portions of the data, referred to as *morsels*. These pipelines are not necessarily independent, as they can share access to data structures. As a result, operators need to be aware of parallel execution and ensure synchronization through efficient lock-free mechanisms.

We generalize this idea to heterogeneous CPU-GPU systems by introducing the *heterogeneous morsel-driven parallelism*. In heterogeneous morsel-driven parallelism, GPU streams, just like other CPU threads are also executing pipelines with different morsels of data. Unlike the standard morsel-driven parallelism, the pipelines here are required to be independent. This is because CPUs and GPUs might have substantially different algorithms for implementing the same operators. For example, while most CPU implementations of hash-joins have a build and a probe phase, in some GPU implementations these two phases are not clearly separated. Moreover, maintaining shared data structures like hash tables between a CPU and a GPU might introduce substantial overhead. Thus, in heterogeneous morsel-driven parallelism, the pipelines executed by GPUs must be independent from pipelines executed by CPUs.

Figure 6, shows possible executions of a query plan with 3 pipelines using both standard and heterogeneous morsel-driven parallelism. Since pipelines P_1 and P_2 are independent, they can be executed on different devices. Multiple CPU threads can process different morsels of pipeline P_1, while different GPU streams can process different morsels of pipeline P_2. In some cases, this can provide speedup even when the CPU↔GPU data transfers are taken into account. For example, when the pipeline P_1 is much heavier than the pipeline P_2, the GPU execution of P_2 might be completely overlapped with the CPU execution of the pipeline P_1, including the data copies. We expect the heterogeneous morsel-driven parallelism to provide more significant speedups with newer devices with higher CPU-GPU interconnect bandwidth, like the NVIDIA's Grace-Hopper series [41] or AMD's MI300 series [2].

5.2 Communication and Computation Overlap

The previous section shows how communication and computation can be overlapped by executing independent pipelines on CPUs and GPUs. However, overlapping communication and computation is also possible within the same pipeline, where some operators are executed on a CPU and some on a GPU. For example, the pipeline P_3, from the previous section (Figure 6) might be a good candidate to execute partially on a CPU and partially on a GPU.

In practice, it is common for pipelines to start with filter and project operators, followed by heavier operators like hash-joins. In such cases, it might make sense to execute the filter and project operators on a CPU and offload the heavy operators like hash-joins to a GPU. If the filter is selective, it can reduce the amount of data that needs to be transferred to a GPU.

These data transfer costs can further be hidden by overlapping them with the computations, as shown in Figure 7. The CPU operators can keep processing other batches while the data is asynchronously being copied to a GPU. Once the data is transferred, the GPU computation can further overlap with the CPU computation, which can result in significant speedups. In practice, this is often achieved by creating a stream for CPU→GPU copies (H2D), another stream for GPU→CPU copies (D2H) and multiple streams for kernel execution (Kernel), as shown in Figure 7. This setting makes sense since data transfers in the same direction cannot overlap.

6 Performance Evaluation

6.1 Experimental Setup

6.1.1 Workload. We used the full TPC-H benchmark with the scaling factor 10 for all experiments. Each query was run 10 times, and the best time is reported for each query engine. In all experiments, the initial data and the final result of each query are stored on the CPU, with all timings including all CPU↔GPU data transfers. All data tables are initially stored as CSV files, and the time of loading the tables from the file system is not included in the reported timings.

6.1.2 Query Engines. We compare Maximus `v0.1` to other state-of-the-art CPU and GPU query engines, including DuckDB (CPU, v1.1.1) [46], DataFusion (CPU, `v42.0.0`) [30], Acero (CPU, Apache Arrow `v17.0.0`) [15], BOSS (CPU+GPU, git commit `15d027e`) [37], HeavyDB (GPU, the open-source `v7.1.0`) [20] and Polars (GPU, `v1.9.0`) [45]. Maximus `v0.1` was built with Apache Arrow `v17.0.0` [15] and cuDF [55] `v24.08.03`.

6.1.3 Hardware. All experiments are performed on an `a2-highgpu-1g` VM instance on Google Cloud Platform, with a single `Intel(R) Xeon(R) CPU @ 2.20GHz` CPU with 85GB RAM, 6 physical and 12 virtual cores, and a single `NVIDIA A100` with 40GB GPU memory. The bandwidth between a CPU and a GPU is 12.35GB/s, as measured by NVIDIA's `nvbandwidth` [10] library, which corresponds to a `PCI` link version 3.0. The OS version is `Ubuntu 24.04.1 LTS`, with the Linux kernel version `6.8.0-1015-gcp`. The NVIDIA driver version is `555.58.02` and the CUDA version is 12.5. The main compiler used is `GNU GCC v13.2.0`.

For easier comparison and visualization, some of the results in this section are normalized, which is clearly stated in each experiment.

6.2 The Full TPC-H Benchmark

We ran the full, end-to-end TPC-H benchmark (SF=10) for DuckDB (CPU), DataFusion (CPU), Apache Acero (CPU), BOSS (CPU+GPU), HeavyDB (GPU) and Polars with the cuDF backend (GPU). For Maximus, we ran both the CPU version using the Apache Acero operators, as well as the GPU version using the cuDF operators. Both Maximus backends also include some of the Maximus native operators, like the LocalBroadcast operator (see Section 3.5.1). The initial data and the final

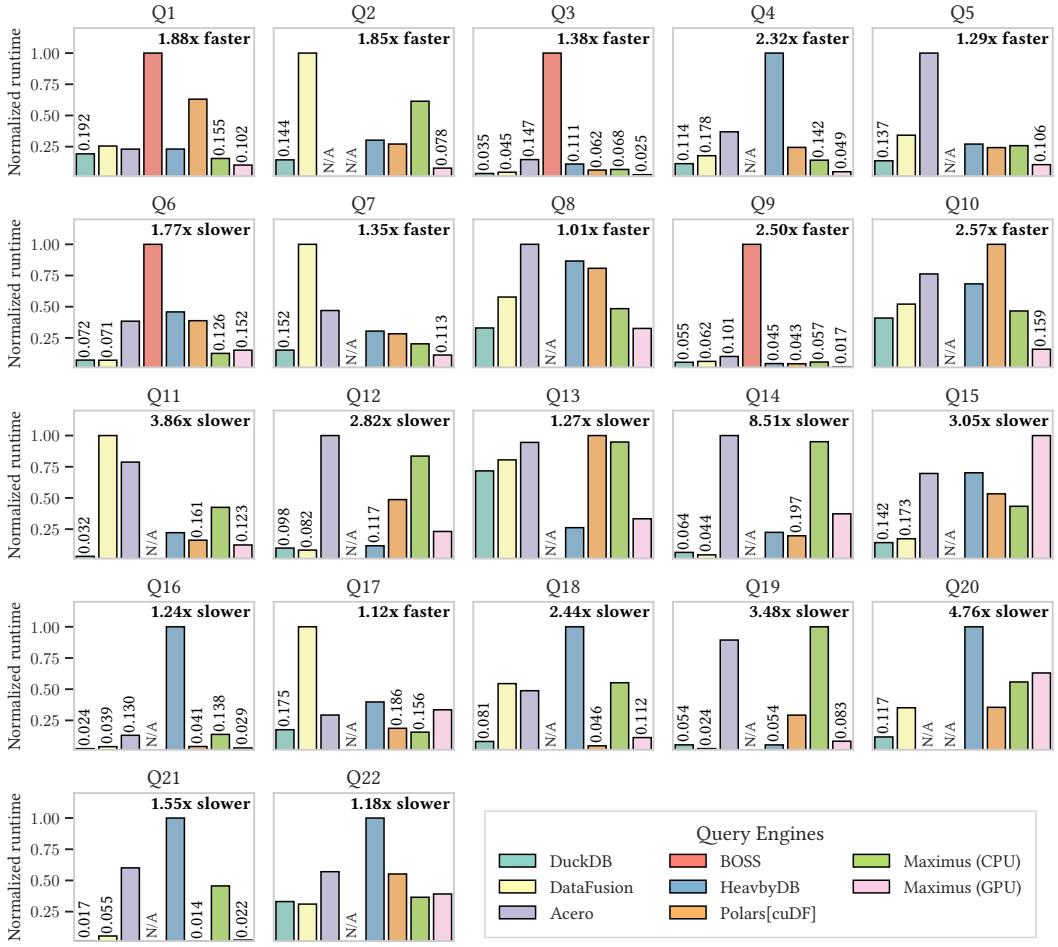


Fig. 8. The execution times for all TPC-H queries for different query engines. All queries start and end on the CPU and all runtimes include all CPU↔GPU data transfer times. The runtimes are normalized by the maximum time for each query. To improve visibility, additional annotations have been added to bars with low height. The upper right corner in each subplot shows the speedup of the best Maximus time (CPU, GPU) vs. the best of other engines.

data resides on the CPU, with all the data transfer times CPU↔GPU being included in the reported timings. The results are summarized in Figure 8. The upper right corner of each subplot represents the speedup of the best Maximus time (CPU, GPU) vs. SOTA, i.e. the best time of other query engines. For each query, all timings are normalized by the maximum execution time among all engines for that query.

The results indicate that Maximus is up to almost 4x faster than the best state-of-the-art engines in some cases. It is important to note that these results are achieved in a setting which favors the CPU-engines, since the tables start and end on the CPU for each query, and the timings include data transfers overhead. Moreover, the Maximus CPU version is mostly using the Acero operators, which are up to 35x slower than DuckDB in some cases. The results where Maximus is slower than

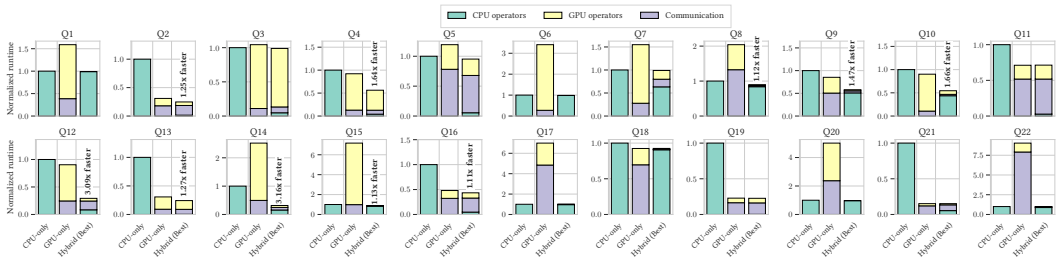


Fig. 9. The comparison of different execution policies in Maximus for TPC-H queries. The values are normalized by the CPU-only time within each query. Wherever the hybrid version was faster or slower by more than 5% vs. the best Maximus (CPU-only, GPU-only) version, the ratio is explicitly annotated above the bar corresponding to the hybrid execution time.

SOTA also indicate that Maximus query plans for these queries are not as optimized as, e.g., in DuckDB.

When compared to BOSS, Maximus is up to 18x faster on the CPU and up to 58x faster on the GPU, in some cases, indicating the performance advantage of the operator-level integration, as a low-level type of integration, vs. the partial query evaluation concept used in BOSS.

6.3 The Communication-Computation Overlap

To measure the effect of overlapping communication and computation on performance, we compared the timings of the CPU-only, the GPU-only and the hybrid execution policies in Maximus. The hybrid execution policy usually performs the initial filtering and projections on the CPU and offloads other operators to the GPU, if the data to be transferred is not too large. The hybrid execution policy aims to pick the best deployment, which also includes running only on the CPU or only on the GPU.

All tables are initially stored in the CPU RAM memory using the arrow format. We notice that in cases when each table is initially stored in multiple chunks, the hybrid execution was especially beneficial. In such cases, the data transfer cost to GPU is affected by higher CPU-GPU data transfer latency, making the CPU more efficient in performing the initial operators like filtering and projections on the data. While the CPU is processing the new batches, the processed batches are repackaged into pinned memory using the pinned memory pool and the asynchronous CPU→GPU copies are initiated (see Section 5.2).

We compared the CPU-only, the GPU-only and the simple hybrid execution policies, measuring the time spent in CPU operators, GPU operators and the CPU↔GPU data transfers. The results are summarized in Figure 9. For each query, the runtimes are normalized by the total execution time of the CPU version for that query. The speedups of the hybrid version vs. the second best is also annotated in the figure. The N/A entries in the figure indicate that the corresponding engine either failed to run the query or does not support it. The results show that the hybrid version can be more than 3x faster than both the CPU-only and the GPU-only versions in some cases, such as Q12 and Q14. In these queries, the initial filtering, which is performed on the CPU, reduces the amount of data, so that the cost of data transfers to the GPU is reduced, allowing the heavier operators like hash-join and group-by to be performed more efficiently on the GPU.

However, there are queries where the communication cost is too large to allow efficiently offloading any part of the query plan execution to the GPU. Examples of such queries are Q1 and Q6. In Q1, the projection operator computes multiple expressions, expanding the data with additional columns, before performing grouped aggregations. This means that offloading the execution to

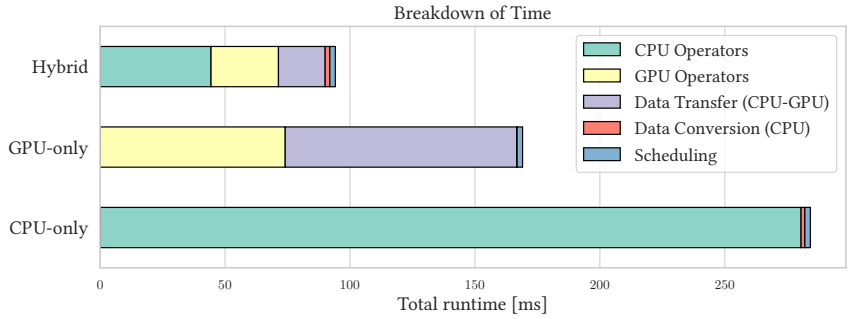


Fig. 10. The breakdown of the overall time for executing all 22 TPC-H queries, using different execution policies.

the GPU at that point would require transferring more than 1GB of data, thus outweighing the performance benefits of utilizing the GPU. For these reasons, the hybrid version offloads the execution to the GPU, only if the data to be transferred is reasonably small, which explains why the CPU-only and hybrid execution times for these queries are the same.

These results provide interesting insights, which can be used for determining the job placement in heterogeneous systems. For example, high-selectivity filters followed by heavy joins might benefit from the CPU-GPU co-processing.

6.4 Breakdown of Time

To assess the overheads of the Maximus framework, we ran the full TPC-H benchmark using the CPU-only, GPU-only and Hybrid execution policies in Maximus. We measured the time spent in CPU operators, GPU operators, data conversions, CPU↔GPU data transfers and scheduling. The full TPC-H benchmark was run 10 times, and the cumulative time spent in each profiling region, over all 22 TPC-H queries, across the 10 runs is presented in Figure 10.

The figure shows that the GPU-only version is 1.72x faster than the CPU-only version when all TPC-H queries are run. Similarly, the hybrid CPU-GPU version is almost 3x faster than the CPU-only version and 1.7x faster than the GPU-only version.

The CPU↔GPU data transfer is by far the largest overhead, taking 55% of the total time for the GPU-only version and 20% of the total time for the hybrid version. It is interesting to note that without the data transfer overhead, the hybrid and the GPU-only versions would have similar runtimes, indicating that fast interconnects might favor the GPU-only version. However, faster interconnects would also allow the hybrid version to explore more interesting operators placements heuristics, which could in turn further reduce the time spent in operators with hybrid execution, in addition to reducing the data transfer time.

The data conversion takes 0.5% of the total execution time for the CPU-only version, <0.1% of of time for the GPU-only version and 2% for the hybrid version. This includes the time spent in the conversion between `acero::compute::ExecBatchh` used by the Apache Acero operators and the standard arrow formats like `arrow::compute::RecordBatchh` and `arrow::compute::Tablee`. As described in Section 3.3, Maximus is using highly efficient data conversions between various formats that are zero-copy and lazy, whenever possible, which minimizes this overhead.

The scheduling overhead is less than 0.5% in all cases. This overhead mostly includes splitting a query plan into operator pipelines, topologically sorting the dependency graph of tasks and scheduling the multithreaded execution of tasks. In the TPC-H benchmark, the query plans consist

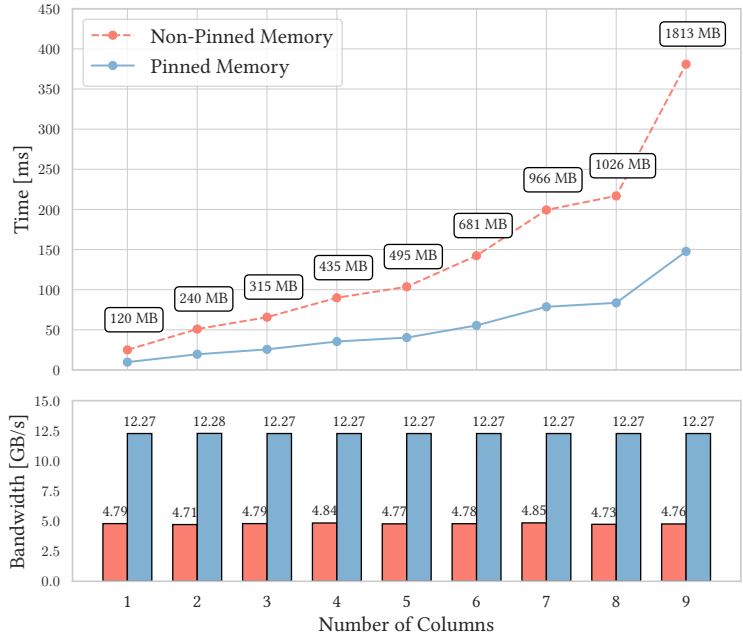


Fig. 11. The efficiency of transferring the *Orders* table from CPU→GPU when it is stored in pinned and non-pinned CPU memory regions. The upper plot shows the total data transfer time and the amount of transferred memory, whereas the bottom plot shows the achieved interconnect bandwidth in both scenarios.

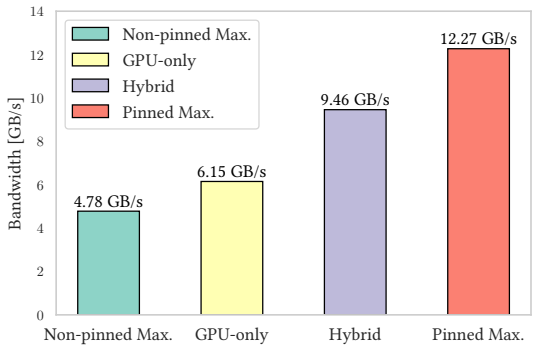


Fig. 12. The comparison of the achieved CPU↔GPU interconnect (PCIe 3.0) bandwidth. The non-pinned and pinned maximum bandwidths are the average bandwidths achieved in isolation whereas the GPU-only and the hybrid are the interconnect bandwidths achieved when running the full, end-to-end, TPC-H benchmark.

of <50 operators, which makes these costs negligible. Since the data transfer cost is the most dominant overhead of execution, we further investigate its efficiency in Section 6.5.

6.5 Data Transfer Efficiency

We assess the CPU↔GPU data transfer efficiency by measuring the achieved CPU-GPU interconnect (PCIe 3.0) bandwidth in isolation through microbenchmarks, and then with the end-to-end TPC-H benchmark.

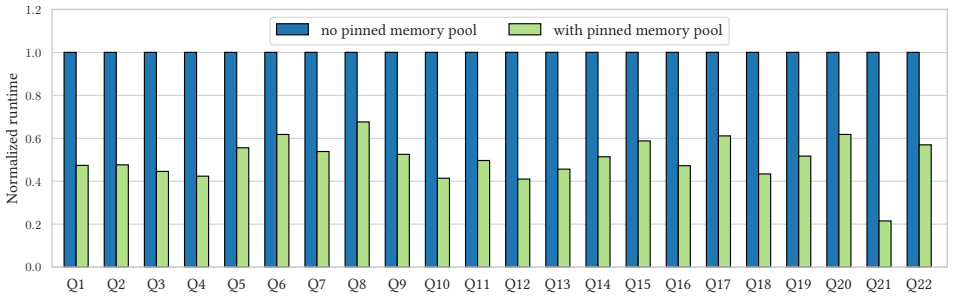


Fig. 13. The comparison of the Maximus GPU performance with and without using the pinned memory. The runtimes are normalized with respect to the runtime without using the pinned memory pool.

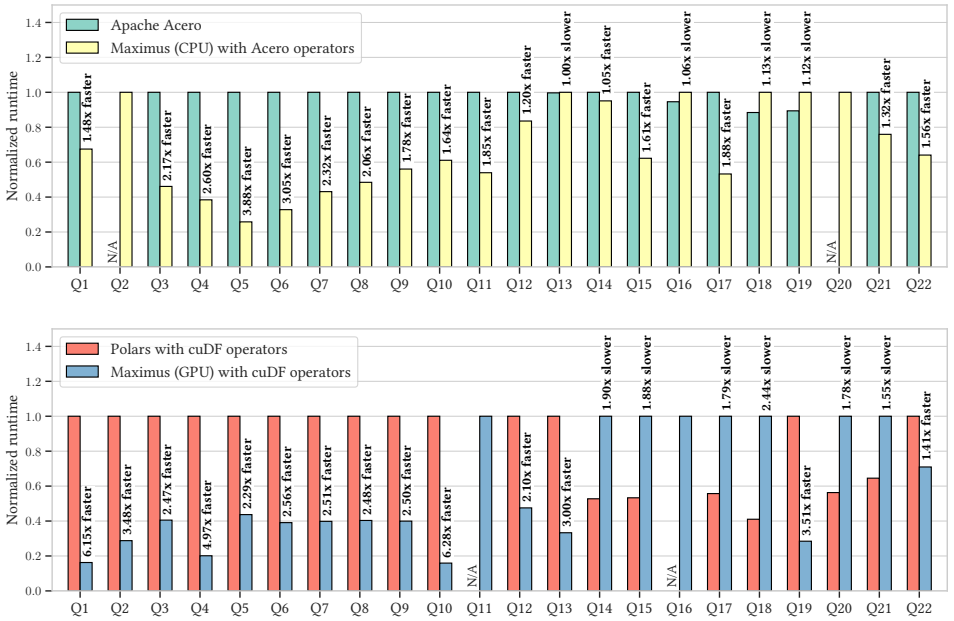


Fig. 14. A comparison of Maximus with third-party operator vs. the (native) engines of these operators. For each query, the runtimes are normalized by the maximum time for that query.

6.5.1 Microbenchmarks. We assessed the bandwidth of transferring the table *Orders*, which is part of the TPC-H benchmark, from the CPU to the GPU in isolation. This table consists of 9 columns: 3 of type `int64`, 1 of type `float64`, 1 of type `date32`, and 4 of type `string`. We measured the bandwidth incrementally, starting with measuring the bandwidth when only the first column is transferred to the GPU, then when the first two columns are transferred, and continuing in this manner up to transferring all 9 columns. It is important to note that variable-length types, such as `string`, require transferring the offset arrays in addition to the column values. For instance, transferring all 9 columns involves sending 13 messages, accounting for the offset arrays associated with each `string` column. The benchmark was performed in two scenarios: 1) when the table is initially stored in a CPU memory region which is not pinned and 2) in a pinned CPU memory region.

Figure 11 shows the time spent in data transfers and the amount of transferred memory in both scenarios (pinned and non-pinned memory) as well as the achieved interconnect bandwidth. We can observe that the bandwidth is stable across various types and number of columns. When the table to be transferred is residing in pinned memory, the data transfers achieve on average 12.27 GB/s, which is 99% of the max. bandwidth measured by *nvbandwidth* (see Section 6.1.3). When the table is residing in non-pinned memory, it achieves on average 4.78 GB/s, which is 38% of the max. bandwidth measured.

6.5.2 End-to-End Benchmarks. Although pinned memory offers higher data transfer bandwidth on PCIe interconnects, it is considered a scarce resource and should not be overused [42]. For this reason, Maximus uses a memory pool of pinned memory, where the maximum size of the memory pool is a tunable parameter. Due to the limited amount of pinned memory, it is interesting to see what bandwidth can be achieved in an end-to-end benchmark.

To this end, we measured the bandwidth achieved when the full TPC-H benchmark is performed using GPU-only and Hybrid execution policies in Maximus. This is compared to the average bandwidths achieved in microbenchmarks with pinned (12.27 GB/s) and non-pinned memory (4.78 GB/s). Figure 12 shows that the GPU-only version of Maximus achieves the interconnect bandwidth of 6.15 GB/s, which is almost 1.3x higher than the non-pinned bandwidth achieved in isolation, but 2x lower than the max bandwidth achieved with pinned memory in isolation. The hybrid version achieved a bandwidth of 9.46 GB/s, which is almost 2x higher and 1.3x lower than the max. non-pinned and pinned bandwidths in isolation, respectively. This is reasonable, since the amount of pinned memory was limited.

In addition to the achieved bandwidth, we also measured the speedup achieved when using the pinned memory pool. Figure 13 compares the GPU-only version of Maximus with and without using the pinned memory pool, for each TPC-H query. The results are normalized with respect to the runtime without using the memory pool. The results indicate that using the pinned memory improves the performance around 2x in most queries. Higher speedups are generally achieved on queries that move more data. This is expected, as the CUDA copies do not have to go through intermediary buffers of pinned memory, but can be performed directly.

6.6 Maximus vs. Native Engines

The operator-level integration enables the integrated operators to take advantage of the advanced features that Maximus is providing, such as the overlap of communication and computation, which is especially important in heterogeneous environments with hybrid execution. However, even the CPU-only and the GPU-only execution can benefit from the efficient pipelining and the parallel execution of independent pipelines, provided by the Maximus executor. Moreover, integrating third-party operators into Maximus allows additional tuning of parameters, such as the number of threads, the processing chunk size, the amount of pinned memory to be used, and others, which might not be controllable in native engines. To measure the efficiency of the Maximus executor, we compared the performance of the Maximus CPU backend (using the Acero operators) with the native Acero engine. Similarly, we also compared the Maximus GPU backend (using the cuDF operators) with the Polars[cuDF] engine, that is also using the cuDF operators. Since cuDF is not a fully-fledged engine, we used Polars as an alternative engine using the same GPU operators as Maximus. The results are summarized in Figure 14. The N/A entries indicate that the corresponding engine either failed to run the query or does not support it.

6.6.1 Maximus (CPU) vs. Apache Acero. The results in Figure 14 show that Maximus (CPU) with the Acero operators outperforms the Acero engine in almost all TPC-H queries, achieving the speedup of almost 4x, in some queries. These results are the consequence of some of the design

differences between Maximus and Acero engines. Both Acero and Maximus are built around morsel-driven parallelism, which is sensitive to the processing chunk size and the cache size. In Acero, the processing chunk size does not seem to be automatically fine-tuned, although it significantly impacts performance [16]. For example, when running TPC-H Q1, the initial acero filter operator in a query plan takes 154 ms if the table is initially stored with chunk size 2^{20}, and 421 ms with chunk size 2^{30}, both of which are often used in practice. This applies not only to the initial results, but also to the intermediate results.

In Maximus, on the other hand, it is possible to control almost any aspect of the execution, thanks to the operator-level integration, since the operators are integrated at the physical level. In addition to this, Maximus also has an additional level of parallelism, where independent pipelines can be executed in parallel, with the possibility to control the number of outer and inner threads (see Section 4.5), which is not possible in Acero. This allows Maximus to outperform Acero using its own operators.

6.6.2 Maximus (GPU) vs. Polars (cuDF). The results in Figure 14 show that Maximus (GPU) achieves a speedup of up to 6x when compared to Polars[cuDF], in some cases. There are two important differences, between Maximus and Polars, affecting the performance. First, Maximus achieves higher CPU↔GPU interconnect (PCIe 3.0) bandwidth in data transfers due to the efficient use of pinned memory, which is a tunable parameter in Maximus. Second, Polars generally tends to process the data in smaller chunks than Maximus. However, this also can make Polars faster in queries with early stopping, such as, e.g. queries which contain a LIMIT operator. Efficiently handling such queries is future work.

7 Conclusions

As hardware continues to evolve, the increasing prevalence of accelerators, such as GPUs, necessitates that query engines adapt to efficiently leverage modern hardware. This evolution has led to a proliferation of specialized query engines, each tailored for specific workloads and hardware configurations. The resulting diversity leads to engines that are efficient on the target hardware but often overly specialized, making it necessary to reimplement operators when the hardware changes or evolves.

To address these challenges, we have developed Maximus – a novel, unified query engine that can execute arbitrary hybrid CPU-GPU query plans thanks to its highly modular and composable design. Through the concept of operator-level integration, Maximus can reuse the existing operators from third-party engines without the need to reimplement them, while preserving the full control over the query execution. Through a series of benchmarks, we have demonstrated that using third-party operators in Maximus, not only has no overhead, but can even be significantly faster than when using the same operators within their native engines. Moreover, the operator-level integration, coupled with Maximus modular design, allows implementing various mechanisms for efficiently utilizing the CPU-GPU parallelism and for overlapping communication with computation. This can further enhance the performance, especially in heterogeneous systems with both CPUs and GPUs. By open-sourcing Maximus, we invite the community to integrate operators from other state-of-the engines like DuckDB or Velox. In the future, we want to extend Maximus to also work with FPGAs and other hardware accelerators, such as smart NICs, DPUs, and computational storage.

Acknowledgments

The authors thank Bowen Wu and Jonas Dann at ETH Zurich for their invaluable feedback on Maximus and their help with the further design of the system.

References

[1] Rana Alotaibi, Yuanyuan Tian, Stefan Grafberger, Jesús Camacho-Rodríguez, Nicolas Bruno, Brian Kroth, Sergiy Matusevych, Ashvin Agrawal, Mahesh Behera, Ashit Gosalia, Cesar Galindo-Legaria, Milind Joshi, Milan Potocnik, Beysim Sezgin, Xiaoyu Li, and Carlo Curino. 2024. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All? arXiv:2411.13704 [cs.DB] https://arxiv.org/abs/2411.13704

[2] AMD. [n.d.]. AMD Instinct MI300 Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300.html. Accessed: 2024-10-17.

[3] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 94–103. doi:10.1145/1735688.1735706

[4] Maximilian Bandle and Jana Giceva. 2021. Database technology for the masses: sub-operators as first-class entities. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2483–2490. doi:10.14778/3476249.3476296

[5] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 47, 11 pages.

[6] Sebastian Breß. 2014. The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (11 2014), 199–209. doi:10.1007/s13222-014-0164-z

[7] Nicolas Bruno, César Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 18–30. doi:10.1145/3626246.3653369

[8] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. doi:10.14778/3632093.3632107

[9] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. doi:10.14778/3303753.3303760

[10] NVIDIA Corporation. [n.d.]. nvBandwidth. https://github.com/NVIDIA/nvbandwidth. Accessed: 01.10.2024.

[11] Bill Dally. 2011. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 878–878. doi:10.1109/IPDPS.2011.420

[12] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (June 2020), 48–57. doi:10.1145/3361682

[13] Christopher Dennl, Daniel Ziener, and Jurgen Teich. 2012. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM '12)*. IEEE Computer Society, USA, 45–52. doi:10.1109/FCCM.2012.18

[14] Jens Dittrich and Joris Nix. 2019. The Case for Deep Query Optimisation. *ArXiv* abs/1908.08341 (2019). https://api.semanticscholar.org/CorpusID:201314319

[15] The Apache Software Foundation. [n.d.]. Acero: A C++ streaming execution engine. https://arrow.apache.org/docs/cpp/streaming_execution.html. Accessed: 2024-10-14.

[16] The Apache Software Foundation. [n.d.]. Apache Arrow Acero Developer's Guide. https://arrow.apache.org/docs/cpp/acero/developer_guide.html#batch-size. Accessed: 2024-10-14.

[17] The Apache Software Foundation. [n.d.]. Apache Gluten (incubating). https://github.com/apache/incubator-gluten. Accessed: 01.10.2024.

[18] Stella Giannakopoulou, Manos Karpathiotakis, Benjamin Gaidioz, and Anastasia Ailamaki. 2017. CleanM: an optimizable query language for unified scale-out data cleaning. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1466–1477. doi:10.14778/3137628.3137654

[19] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: a main memory hybrid storage engine. *Proc. VLDB Endow.* 4, 2 (nov 2010), 105–116. doi:10.14778/1921071.1921077

[20] Heavy.AI. [n.d.]. HeavyDB. https://github.com/heavyai/heavydb. Accessed: 2024-10-14.

[21] Pedro Holanda and Hannes Mühleisen. 2019. Relational Queries with a Tensor Processing Unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 3 pages. doi:10.1145/3329785.3329932

[22] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distrib. Syst.* 33, 6 (June 2022), 1303–1320. doi:10.1109/TPDS.2021.3104255

[23] Wenqi Jiang, Dario Korolija, and Gustavo Alonso. 2023. Data Processing with FPGAs on Modern Architectures. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 77–82. doi:10.1145/3555041.3589410

[24] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (July 2023), 3461–3474. doi:10.14778/3611479.3611539

[25] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. doi:10.14778/3551793.3551801

[26] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1001–1013. doi:10.1145/3448016.3457288

[27] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. The SABER system for window-based hybrid stream processing with GPGPUs: demo. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems* (Irvine, California) *(DEBS '16)*. Association for Computing Machinery, New York, NY, USA, 354–357. doi:10.1145/2933267.2933291

[28] Dario Korolija, Dimitrios Koutsoukos, K. Keeton, Konstantin Taranov, Dejan Milojicic, and Gustavo Alonso. 2021. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. *ArXiv* abs/2106.07102 (2021). https://api.semanticscholar.org/CorpusID:235421855

[29] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: modular relational analytics over heterogeneous distributed platforms. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3308–3321. doi:10.14778/3484224.3484229

[30] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 5–17. doi:10.1145/3626246.3653368

[31] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. doi:10.1145/2588555.2610507

[32] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast in-memory SQL analytics on typed graphs. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 265–276. doi:10.14778/3021924.3021941

[33] Alec Lu, Jahanvi Narendra Agrawal, and Zhenman Fang. 2024. SQL2FPGA: Automated Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 39 (Sept. 2024), 28 pages. doi:10.1145/3674843

[34] Jimmy Lu. 2024. Techniques in Accelerating Query Processing on GPU. In *Proceedings of the 16th Workshop on Computational Data Management Systems (CDMS) (CEUR Workshop Proceedings, Vol. 3462)*. https://ceur-ws.org/Vol-3462/CDMS16.pdf

[35] Nantia Makrynioti, Ruy Ley-Wild, and Vasilis Vassalos. 2021. Machine learning in SQL by translation to TensorFlow. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning* (Virtual Event, China) *(DEEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. doi:10.1145/3462462.3468879

[36] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. 2021. Resource-Efficient Database Query Processing on FPGAs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware* (Virtual Event, China) *(DAMON '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. doi:10.1145/3465998.3466006

[37] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2024. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow.* 17, 4 (mar 2024), 877–890. doi:10.14778/3636218.3636239

[38] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. doi:10.1145/3399666.3399911

[39] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. doi:10.14778/2002938.2002940

[40] nsight. [n.d.]. Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2024-10-14.

[41] NVIDIA. [n.d.]. NVIDIA GH200 Grace Hopper Superchip. https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip. Accessed: 2024-10-17.

[42] NVIDIA Corporation. [n.d.]. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#pinned-memory Section: Pinned Memory, Accessed: 2024-10-14.

[43] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384. doi:10.14778/3554821.3554829

[44] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (jun 2023), 2679–2685. doi:10.14778/3603581.3603604

[45] Polars. [n.d.]. Polars: Fast DataFrame library. https://github.com/pola-rs/polars. Accessed: 2024-10-14.

[46] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. doi:10.1145/3299869.3320212

[47] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *Proceedings of the Conference on Innovative Data Systems Research*.

[48] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (Jan. 2022), 38 pages. doi:10.1145/3485126

[49] Maximilian Schule, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Gunnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management* (Tampa, FL, USA) *(SSDBM '21)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/3468791.3468840

[50] 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)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. doi:10.1145/3318464.3380595

[51] 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)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. doi:10.1145/3318464.3380595

[52] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E. Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings, Vol. 3462)*, Rajesh Bordawekar, Cinzia Cappiello, Vasilis Efthymiou, Lisa Ehrlinger, Vijay Gadepally, Sainyam Galhotra, Sandra Geisler, Sven Groppe, Le Gruenwald, Alon Y. Halevy, Hazar Harmouch, Oktie Hassanzadeh, Ihab F. Ilyas, Ernesto Jiménez-Ruiz, Sanjay Krishnan, Tirthankar Lahiri, Guoliang Li, Jiaheng Lu, Wolfgang Mauerer, Umar Farooq Minhas, Felix Naumann, M. Tamer Özsu, El Kindi Rezig, Kavitha Srinivas, Michael Stonebraker, Satyanarayana R. Valluri, Maria-Esther Vidal, Haixun Wang, Jiannan Wang, Yingjun Wu, Xun Xue, Mohamed Zaït, and Kai Zeng (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-3462/CDMS8.pdf

[53] substrait io. 2021. *Substrait: Cross-Language Serialization for Relational Algebra.* https://github.com/substrait-io/substrait

[54] Umar Syed and Sergei Vassilvitskii. 2017. SQML: large-scale in-database machine learning with pure SQL. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 659. doi:10.1145/3127479.3132746

[55] RAPIDS Development Team. 2023. *RAPIDS: Libraries for End to End GPU Data Science.* https://rapids.ai

[56] The Apache Software Foundation. [n.d.]. Apache Iceberg. https://iceberg.apache.org/ Accessed: 2025-01-24.

[57] Transaction Processing Performance Council (TPC). [n.d.]. TPC Benchmark™ H (TPC-H). https://www.tpc.org/tpch/. Accessed: 2024-10-14.

[58] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. doi:10.14778/2536206.2536210

[59] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017). https://api.semanticscholar.org/CorpusID:13190998

[60] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1042–1057. doi:10.1145/3470496.3533044

Received October 2024; revised January 2025; accepted February 2025