Understanding the Performance of Native Execution in Big Data Engines: The Good, the Bad, and How to Fix It

Haikai Zhao
Simon Fraser University
Burnaby, BC, Canada
hza214@sfu.ca

Zhenman Fang
Simon Fraser University
Burnaby, BC, Canada
University of Minnesota
Twin Cities, MN, USA
zhenman@sfu.ca

Abstract

Recently, big data engines have increasingly shifted towards offloading compute-intensive workloads from JVM to native execution engines implemented in C++ or Rust. This transition enables better exploitation of modern CPU features, particularly SIMD instructions, to address critical performance bottlenecks. In this paper, we systematically evaluate the performance across three different native vectorized execution engines, including C++ based Velox and ClickHouse, and Rust based DataFusion, for the widely used big data engine Apache Spark. Interestingly, we observe that offloading computation to a native engine does not always guarantee superior performance. Our goal is to provide insights into when to use a native execution engine and which engine to choose, when not to use, and how to automate this decision process.

To achieve this goal, we first investigate the root cause of performance degradation: *the overhead of vector materialization across operators.* Based on the insights, we propose a cost model-driven operator placement strategy that adaptively schedules operators between the native vectorized engine and the JVM code generation engine. Our evaluation demonstrates that this strategy effectively validates our idea and achieves up to 2.1× speedup over always offloading all operators to the native engine.

The transition to native engines also opens new opportunities for hardware acceleration to further improve performance. We first identify the bottlenecks that persist after native execution, and then conduct a case study to accelerate Apache ORC table scan using GPU and FPGA hardware solutions. Our results reveal that FPGA can outperform GPU and deliver performance gains of up to 1.8×.

Keywords

Apache Spark, Native Execution, Hardware Acceleration, Big Data

1 Introduction

With rapid advancements in I/O technologies, including the adoption of SSDs and high-bandwidth network interfaces, the CPU has become the major bottleneck in big data analytics. A common optimization is to enable SIMD vectorization to better leverage the CPU instruction pipeline and cache memory hierarchy [3]. However, today's big data engines, such as Apache Spark [43], Apache Flink [12], and Presto [45], are predominantly JVM-based and largely rely on row-oriented execution models. Such execution models hinder the exploitation of modern CPU features, especially SIMD vectorization. On the other hand, while offloading computation to heterogeneous accelerators presents an alternative strategy to address CPU bottlenecks, JVM also faces challenges in leveraging hardware accelerators due to limited hardware ecosystem support [8, 14, 16], substantial data communication overhead [9, 36], and suboptimal memory management [9].

The introduction of native execution [3] has motivated recent research [21, 56] to offload big data workloads (i.e., operators) to native execution engines developed in C++ and Rust, which are designed to take advantage of SIMD capabilities on modern CPUs. Within the Apache Spark community—one of the most widely used big data engines—the trend has also emerged towards integrating open-source native execution engines, including C++ based Velox [34] and ClickHouse [39], and Rust based Apache DataFusion [20], through the Spark accelerator plugins such as Apache Gluten [40], Auron [2], and Apache DataFusion Comet [7].

Motivation. While prior work [20, 34, 39] has evaluated the performance of each native engine individually, there lacks an in-depth performance comparison between these native engines in a unified setting. Our quick comparison, using the top 20 longest-running TPC-DS queries at a scale factor of 1TB, has revealed two interesting observations: 1) for different queries, different native engines excel; 2) for certain queries, offloading to native engines actually degrades the query performance. Based on these observations, this paper aims to answer the following key questions (summarized in Figure 1). First, when should we choose native execution, when not, and why? Second, if we decide to choose native execution, which native engine should we choose for a given query? Third, after native execution, which operators are the remaining bottlenecks and how much hardware acceleration can potentially help?

Workload Characterization. In this work, we present an in-depth performance characterization of analytic workloads, comparing vanilla Spark (JVM-based) with its various native engine implementations. Our evaluation combines end-to-end TPC-DS benchmarking, quantitative bottleneck analysis of its longest-running queries, and microbenchmarking of critical workloads derived from these bottlenecks. We aim to provide valuable insights into the implementation differences of these engines and highlight potential opportunities for future performance improvements. Notably, the study reveals that universally offloading all workloads to native vectorized engines does not necessarily deliver performance speedups, especially in query pipelines with consecutive join operators, which is observed in Presto and its native execution as well in our ablation study. Here, we notice that in some cases the overhead of intermediate vector materialization between operators can be costly enough to offset native speedups and even cause performance degradation. For such

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.48

workloads, code-generation (code-gen) based execution in JVM is often the optimal solution, as it can fuse operators into nested loops within a single function, allowing data to be kept in registers to avoid data movement overhead. Furthermore, even with SIMD-optimized vectorized execution, we identify that decoding and decompression within columnar table scans remain bottlenecks, highlighting a potential for heterogeneous hardware acceleration. Our analysis extends to a comparative performance evaluation of the widely used Apache ORC [31] table scans across various native engines and hardware accelerator solutions.

Workload Optimization. Building upon the insights gained from workload characterization, we propose two key optimization strategies to enhance the Spark native execution performance: 1) a cost model based operator placement strategy, and 2) native engine integrated with heterogeneous hardware acceleration. Our first strategy addresses performance degradation from vector materialization overhead. Specifically, we propose an adaptive operator placement strategy that employs cost model guided query plan exploration to intelligently select between code-gen and vectorized operator implementations. This approach moves beyond universally offloading to native execution, instead facilitating hybrid execution models. It achieves up to 2.1× speedup and never degrades the performance compared to solely offloading all operators. Our second strategy targets the decoding/decompression bottleneck in table scans, particularly noting the limitations of GPUs for accelerating the scan of columnar formats. Instead, we integrate the native engine with an FPGA-based accelerator, which provides customizable optimizations and a dataflow execution model to pipeline computations and hide data transfer overhead. This integration yields significant performance gains, demonstrating up to 1.8× speedup in the widely used Apache ORC table scan workloads.

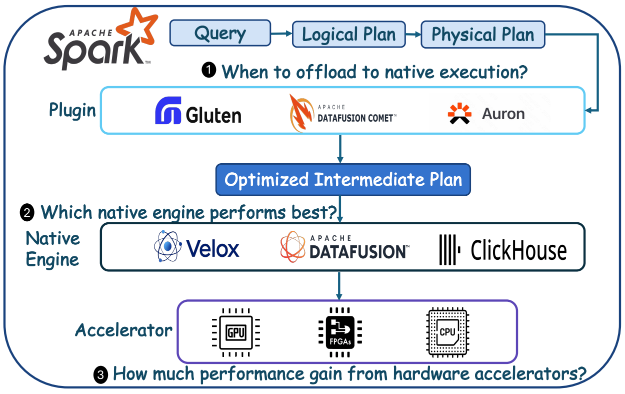
The main **contributions** of this paper are summarized as below:

(1) **Comprehensive Performance Characterization**: We present an extensive comparative evaluation of Apache Spark and its state-of-the-art native execution engines across end-to-end workloads and bottleneck-focused micro-benchmarks.
(2) **Analysis of Tradeoffs for Native Execution**: We present an in-depth comparison of Spark's default code-gen execution and native vectorized execution to determine when and why native engines underperform JVM-based execution.
(3) **Cost Model Guided Operator Placement Strategy**: We propose an adaptive operator placement strategy with cost model guided query plan exploration to dynamically and intelligently select the most efficient operator implementation between code-gen and vectorized execution.
(4) **Exploration of Heterogeneous Hardware Acceleration**: We identify the bottlenecks requiring hardware acceleration, conduct a comparative analysis of native engines and hardware accelerator solutions, and demonstrate the FPGA integration with native engines to offload the table scan bottleneck.

2 Background

2.1 Code-Gen and Native Vectorized Execution

Within Apache Spark's JVM execution engine, the Whole-Stage Code Generation (WSCG) mechanism is employed to fuse multiple nonblocking operators into a single, optimized Java method.



Apache Spark
Query
Logical Plan
Physical Plan
❶ When to offload to native execution?
Plugin
Gluten
Apache DataFusion Comet
Auron
Optimized Intermediate Plan
❷ Which native engine performs best?
Native Engine
Velox
Apache DataFusion
ClickHouse
Accelerator
GPU
FPGAs
CPU
❸ How much performance gain from hardware accelerators?

Figure 1: Apache Spark native execution workflow

This generated code is then compiled into JVM bytecode, which the JIT compiler further optimizes into native machine code. During execution, the native machine code can keep data in CPU registers to avoid data movement overhead [30]. In contrast, most existing native execution engines avoid the JVM overhead and adopt a vectorized execution model on native machines, which extends the classic Volcano-style iterator framework [15]. In the Volcano model, each operator repeatedly invokes a next method to obtain a single tuple from its child operator, incurring substantial interpretation overhead due to frequent function calls. Vectorized execution mitigates this limitation by processing data in batches organized in a columnar layout. The key idea is to choose batch sizes that are sufficiently large to amortize interpretation overhead, yet small enough to remain hot in the CPU cache, thereby avoiding repeated cache-memory boundary crossings and the associated materialization costs [19]. More importantly, it can efficiently leverage the vectorization instructions to boost its performance. In summary, offloading operators from the JVM-based engine to a native engine inherently involves a transition from a data-centric, code-gen execution model to an interpreted, vectorized execution model.

2.2 Apache Spark Native Execution Workflow

Figure 1 illustrates how Apache Spark offloads computational workloads to native execution engines. It requires not only an efficient and extensible query engine library, but also a plugin to serve as the middle layer that is responsible for plan conversion, managing data communication between the JVM and native runtime, and provides a fallback mechanism for operators that are not natively supported. Representative examples of open-source native engines include Velox [34], Apache Arrow DataFusion [20], and ClickHouse [39]. Representative examples of such accelerator plugins include Apache Gluten [40], Datafusion Comet [7], and Auron [2].

The offloading workflow begins with the plugin converting Spark's internally optimized query plan into an intermediate representation (IR), utilizing formats such as Substrait [44] or Protobuf [38]. This IR is subsequently dispatched to the native execution layer via Java Native Interface (JNI) calls, allowing the native engine to build and execute the corresponding query. For efficient data exchange, Apache Arrow [1] serves as the foundational format, facilitating zero-copy data transfers between the different runtime systems. Spark provides Arrow-compatible vectors to the accelerator plugins, which then convert them to the required format of the native engine before passing them to the native runtime via JNI. Upon completion, the native runtime returns results to the JVM in an Arrow-compatible format. Furthermore, some native engines can offload operators to hardware accelerators, offering further performance enhancements.

With the accelerator plugin, Spark retains its distributed control flow, query parser, and optimizer, while offloading query plans and data to native runtimes for execution.

2.3 Apache Spark Native Engines

Given the limitations of the JVM, Databricks was the first to propose offloading Spark computation to its C++ columnar vectorized engine, Photon [3], to further exploit CPU performance with vectorization techniques. However, Photon is proprietary, which has encouraged efforts to integrate alternative, open-source native execution engines with Apache Spark. The following provides an overview of these available open-source native engines (summarized in Figure 1).

Velox. Velox [34] is an open-source native execution engine developed by Meta and implemented as a C++ library. It is designed to integrate seamlessly with existing data management systems. Velox extensively leverages optimization techniques, including vectorized execution, lazy materialization, and adaptive execution.

Apache DataFusion. Apache Arrow DataFusion [20] is an efficient, embeddable, and extensible query execution engine that utilizes Apache Arrow as its in-memory data model. Similar to Velox, it aims to serve as a reusable foundation for building database systems. However, it is written in Rust rather than C++ and offers a comprehensive SQL front-end as well as a planning and optimization framework.

ClickHouse. In contrast to Velox and DataFusion, which are query execution engine libraries intended to be integrated into broader database management systems (DBMS), ClickHouse [39] is a standalone DBMS with both compute and storage runtimes. In this paper, we focus specifically on the query execution engine of ClickHouse. It features a high-performance vectorized engine optimized for real-time analytics on petabyte-scale data, employing techniques such as SIMD optimizations and runtime-adaptive strategies, including dynamic aggregation and join strategies.

2.4 Apache Spark Accelerator Plugins

The following are state-of-the-art open-source Spark-based plugins designed to offload computational workloads to native engines.

Apache Gluten. Apache Gluten [40] is a plugin designed to accelerate Apache Spark by offloading query plans to Velox or ClickHouse backends. It transforms Spark's physical plans into Substrait representations, which are then sent to native libraries via JNI and translated into the corresponding native query plans.

Apache DataFusion Comet. Apache DataFusion Comet [7] is an Apache Spark accelerator built on top of the Rust-based DataFusion engine. Comet is designed to enhance the performance of Spark workloads while leveraging commodity hardware and seamlessly integrating with the Spark ecosystem without requiring any code changes.

Auron. Similarly, the Auron [2] project accelerates Apache Spark by offloading computation-intensive workloads to the Apache DataFusion engine. Its query plan transformation and data communication processes are similar to those of Apache DataFusion Comet.

Table 1: Summary of Apache Spark execution engines and supported accelerator plugins

Engine	Vanilla Spark	Velox	ClickHouse	DataFusion
Language	Scala	C++	C++	Rust
Data Layout	Mostly Row	Columnar	Columnar	Columnar
Execution Model	Code-Gen	Vectorized	Vectorized	Vectorized
Plugin	N/A	Gluten	Gluten	Auron, Comet
Hardware Support	N/A	GPU	None	None

Summary. Table 1 provides a comparative summary of the three open-source Spark native execution engines and their supported accelerator plugins, as well as vanilla Spark.

3 Evaluation and Analysis

In this section, we first present a comprehensive end-to-end performance analysis of the aforementioned Apache Spark native execution engines, based on the widely used TPC-DS benchmark queries [47] with a scale factor of 1TB. Then we further evaluate and analyze the performance of key operators (such as table scan, hash aggregation, hash join, and sorting) identified during the TPC-DS end-to-end profiling, using these native engines.

3.1 Hardware and Software Setup

For the end-to-end TPC-DS performance evaluation in Section 3.2, we set up a Spark cluster of one master node and four worker nodes. Each node is equipped with two 2.2GHz Intel Xeon Silver 4214 CPUs, each CPU with 12 physical cores and 24 hyperthreads, 192GB DDR4 memory, 480GB SSD as the local disk and 10Gb/s NIC. Each CPU core has 32KB L1 cache and 1024KB L2 cache, and all 12 cores share a 16.5MB L3 cache. The system does not include Spectre-related microcode patches or mitigations, and SIMD gather performance is therefore unaffected.

Table 2: Summary of Spark configurations

Configuration	Value
spark.memory.offHeap.enabled	true
spark.memory.offHeap.size	20G
spark.executor.instances	48
spark.executor.cores	4
spark.executor.memoryOverhead	4G
spark.executor.memory	16G
spark.driver.memory	40G

All experiments are conducted using the Spark configurations summarized in Table 2. Other unlisted configurations use the default values. We selected Gluten as the accelerator plugin to evaluate the performance of the Velox and ClickHouse native engines, as it represents the most mature and feature-complete solution available for native engine integration. Auron is used to evaluate the Apache DataFusion engine, as Apache DataFusion Comet is still in its early stages of development and lacks support for several operators in the TPC-DS benchmark queries. All experiments are conducted using Apache Spark 3.5.5, the latest version compatible with the most recent releases of the evaluated native engines (Velox 2025-08-21, ClickHouse 25.8, DataFusion 49.0.0) and accelerator plugins (Gluten v1.5.0, Auron v6.0.0).

3.2 End-to-End Performance Comparison

For the end-to-end query performance comparison, we employ the TPC-DS benchmark suite [47], which includes 99 queries. In our experiments, the benchmark queries are executed with a scale factor (SF) of 1TB, and the table data are generated in Apache Parquet format [33]. We focus our analysis on the 20 longest-running queries, as they collectively account for approximately 83% of the total execution time, while the remaining queries each complete within almost a few seconds. Figure 2 illustrates the results for the top 20 longest-running queries. To isolate query execution time, no full materialization of results is performed. To further analyze and understand the performance differences across native engines under various workloads, we summarize the workload bottlenecks for the evaluated queries in Table 3.

Table 3: Query bottlenecks and their CPU percentages

Bottleneck Operator	**Queries (with operator time%)**
Hash Join	Q14a (44.1%), Q14b (44.6%), Q64 (44.1%), Q72 (84.6%), Q95 (76.0%)
Table Scan	Q9 (34.7%), Q16 (34.7%), Q24a (67.1%), Q24b (66.8%), Q50 (33.1%), Q75 (44.4%), Q88 (74.2%)
High-cardinality Hash Aggregation	Q4 (78.8%), Q11 (44.4%), Q23a (67.6%), Q23b (61.2%), Q28 (74.3%)
Low-cardinality Hash Aggregation	Q67 (74.0%)
Sort	Q78 (56.8%), Q93 (70.2%)

The following observations can be made from Figure 2:

(1) Among the 20 longest-running queries, native execution using Velox, ClickHouse, and DataFusion achieves overall speedups of 1.98×, 1.49×, and 1.45×, respectively. Velox runs fastest on 11 queries, ClickHouse on 6 queries, DataFusion on 1 query, and surprisingly, Vanilla Spark still performs the best on 2 queries.

(2) The effectiveness of native execution is inconsistent for queries where the join operator is the primary bottleneck (e.g., Q14a, Q14b, Q64, Q72, and Q95). While Q14a, Q14b, and Q64 show improved performance for native execution, Q72 and Q95 suffer performance degradation, performing even worse than vanilla Spark. This observation contrasts with earlier workload characterization studies [18], which concluded that vectorized execution excels for join-heavy queries. We investigate the root cause of this discrepancy and present our solutions in Section 4.

(3) The native execution engines prove consistently effective in accelerating table scan workloads. For example, in query Q88, where the table scan accounts for 74.2% of the total query time, the speedups achieved are 1.42× for Velox, 1.52× for ClickHouse, and 1.33× for DataFusion.

(4) Native execution demonstrates substantial performance improvements in queries dominated by hash aggregation. Among the evaluated engines, Velox consistently delivers the best performance across both high-cardinality (e.g., Q4, Q11, Q23a, Q23b and Q28) and low-cardinality (e.g., Q67) aggregation-intensive queries. In contrast, ClickHouse shows the weakest performance, particularly in scenarios involving high-cardinality aggregations.

(5) For queries where the bottleneck is sorting (e.g., Q78 and Q93), native execution delivers significant and consistent performance gains across all engines, with ClickHouse performing the best among them.

Next, we design micro-benchmarks to evaluate each performance-critical operator, including hash join, table scan, hash aggregation and sorting, for all native execution engines, aiming to offer deeper insights into their performance differences. We report the results in Sections 3.3, 3.4, 3.5 and 3.6, all using a single-node Spark cluster with the same configuration as Section 3.2.

Summary of TPC-DS End-to-End Performance: Native execution engines demonstrate significant performance advantages over JVM-based computation, with Velox, ClickHouse, and DataFusion achieving end-to-end analytic workload speedups of 1.98×, 1.49×, and 1.45×, respectively. However, there are exceptions where vanilla Spark performs better, especially for some hash-join heavy queries.

3.3 Hash Join

Hash join consists of two main phases: 1) a build phase that scans the build-side table and constructs an in-memory hash table, and 2) a probe phase that scans the probe-side table, applies the same hash function, and probes the hash table to produce matching tuples. The three native engines evaluated support vectorized hash operations such as batched inserts and lookups. To evaluate in-memory broadcast hash join performance, we compare vanilla Spark to its integration with native vectorized engines on a single-node Spark cluster using a broadcast hash join query between the `store_sales` and the `customer` table. The `customer` table is broadcast to construct an in-memory hash table, enabling efficient vectorized lookups during the probe phase.

```sql
SELECT * FROM customer, store_sales WHERE
    customer.customer_id =
    store_sales.customer_id;
```

Both tables include a single 4-byte integer column called `customer_id`. The probe-side table `store_sales` is fixed at 64 million tuples, while the build-side table (`customer`) is varied such that the hash table size ranges from 8 KB to 256 MB.

Figure 3 shows how hash table size affects the runtime of in-memory hash join operations. Our evaluation demonstrates that Velox achieves the highest average speedup of 2.02× over vanilla Spark, followed closely by ClickHouse at 1.95×, while DataFusion provides a 1.26× improvement. Across all engines, runtime remains stable when the hash table fits within a cache level but increases sharply once it exceeds cache boundaries. Specifically, tables smaller than the L2 cache size (1 MB) reside entirely in L2 cache, ensuring low access latency. When the table size exceeds L2 cache size, L3 cache accesses are required, causing a noticeable step increase. Runtime stabilizes again for tables up to 16 MB, which fit in L3 cache. Beyond L3 cache size, frequent main memory accesses incur high latency, significantly increasing runtime.

The build cost is influenced mainly by one-time materialization, whereas the probe cost escalates sharply once the hash table exceeds cache limits (1 MB for L2, 16 MB for L3) due to frequent

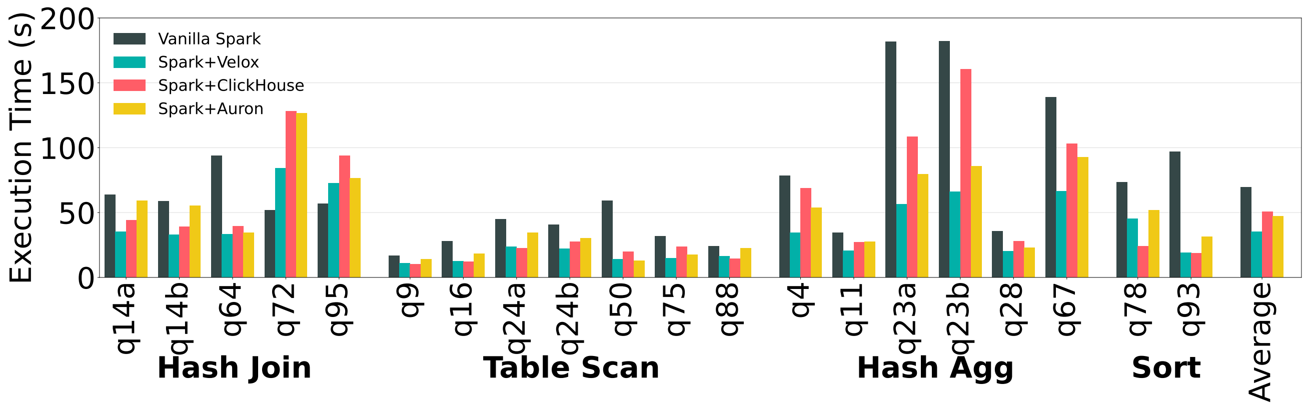


Figure 2: Comparison of engines using 20 longest-running TPC-DS queries with SF=1TB, grouped by the query bottleneck

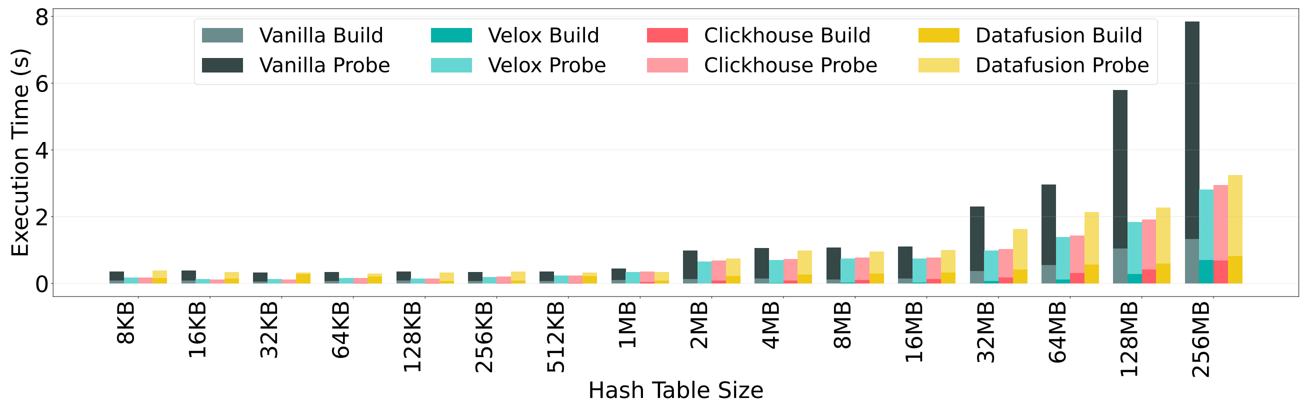
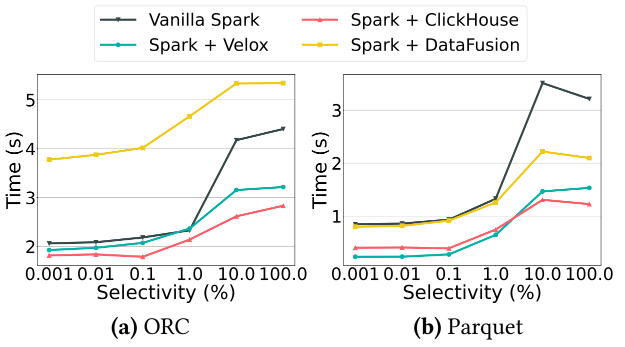


Figure 3: Hash join execution time by engine with build/probe phase breakdown

random DRAM accesses. This makes the probe phase the dominant contributor to total runtime. Furthermore, the performance gap widens beyond the L3 cache as Velox, ClickHouse, and DataFusion exploit SIMD instructions to amortize cache-miss costs, unlike vanilla Spark which processes tuples individually.

Summary of Hash Join Performance: Native vectorized execution outperforms code-gen approaches in hash join operations, particularly for queries with large hash tables, where vectorization mitigates random memory access and cache miss latencies. Velox excels with smaller hash tables, while ClickHouse performs best with larger ones. However, in complex TPC-DS queries with multiple joins (e.g., Q72 and Q95), native execution exhibits performance degradation compared to code-gen based approaches.



(a) ORC **(b)** Parquet

Figure 4: Table scan performance for Apache ORC and Parquet file formats, under different engines

3.4 Table Scan

Next, we evaluate the performance of range scan operations with varying selectivity. Experiments are conducted on ORC [31] and Parquet [33] formats, with selectivity ranging from 0.001% to 100% over a dataset of 100 million integers. The reported time covers predicate evaluation, disk I/O, decompression, and decoding.

Figure 4(a) shows that ClickHouse's ORC reader outperforms Velox by an average of 11.6%, while the DataFusion ORC reader performs even worse than vanilla Spark due to its reliance on the experimental ORC Rust library, which lacks optimizations such as predicate pushdown and Bloom filters.

Figure 4(b) presents Parquet scan results. ClickHouse and Velox are the best: ClickHouse is slightly faster at high selectivity (>10%), while Velox is marginally better at low selectivity. For vanilla Spark, Spark+DataFusion, and Spark+ClickHouse, scan time at 100% selectivity is lower than at 10%, a consequence of Parquet page index optimizations. Full selectivity enables engines to achieve high I/O throughput via sequential page scans, while moderate selectivity induces page index filtering that causes scattered page accesses, resulting in random I/O patterns and additional metadata processing overhead. Spark+Velox avoids this performance drop because its C++ Parquet reader does not exploit page index filtering; instead, all pages are scanned sequentially and predicate filtering is applied during decoding.

(1) **Summary of ORC Scan Performance**: ClickHouse's native reader exhibits superior performance compared to Velox. DataFusion's ORC reader substantially underperforms relative to vanilla Spark due to the absence of optimizations including predicate pushdown and Bloom filter implementation.
(2) **Summary of Parquet Scan Performance**: ClickHouse integration demonstrates superior performance at higher selectivity thresholds (>= 10%), while Velox integration excels at lower selectivity levels.

3.5 Group By Hash Aggregation

This subsection presents a performance evaluation of hash-based aggregation for the COUNT, SUM, and AVG operators. The dataset is stored in Apache Parquet format, with customer_id used as the group-by key. To evaluate the impact of group-by key cardinality on aggregation performance, we vary the cardinality of customer_id from 0.0001% to 100% of the total number of rows. To isolate the performance of the aggregation operations from external factors such as file I/O and deserialization, the dataset is fully loaded into memory prior to execution. Each query is executed three times, and the average execution time is reported. The queries used in this evaluation are as follows:

```
COUNT: SELECT customer_id, COUNT(quantity)
    FROM store_sales GROUP BY customer_id
SUM: SELECT customer_id, SUM(quantity) FROM
    store_sales GROUP BY customer_id
AVG: select customer_id, AVG(quantity) FROM
    store_sales GROUP BY customer_id
```

where customer_id and quantity are integers, and the table contains 100 million tuples. Figure 5 shows the performance results of three aggregation operations. Velox consistently outperforms other engines across all cardinalities. For low-cardinality group-by keys ($\leq$ 10%), Spark+ClickHouse and Spark+DataFusion achieve similar performance for both COUNT and SUM. In high-cardinality cases (100%), Spark+ClickHouse performs worse than Spark+DataFusion.

The performance gap in high-cardinality aggregation is due to the two-phase aggregation strategy used by most engines: a partial aggregation phase groups rows locally and computes intermediate results, followed by a final aggregation phase that consolidates partial results across partitions. When the number of unique groups is small, partial aggregation effectively reduces data volume, reducing shuffle and final aggregation costs. In high-cardinality scenarios, each key has few rows, and partial aggregation cannot significantly reduce data, resulting in nearly identical workloads in both phases. DataFusion and Velox optimize this by skipping the partial phase when it provides little benefit, directly performing final aggregation. ClickHouse lacks this optimization, leading to higher computational overhead and lower performance for high-cardinality aggregation.

Summary of Hash Aggregation Performance: Velox consistently outperforms alternative engines in COUNT, SUM and AVG aggregations for all group-by key cardinalities. For high-cardinality cases, both Velox and DataFusion surpass ClickHouse due to the partial aggregation skipping.

3.6 Sorting

This subsection presents a performance evaluation of sorting operations with different engines under various scenarios. The dataset used is the store_sales table from the TPC-DS benchmark, stored in Apache Parquet format with the scale factor of 100GB. We execute five queries with different sorting workload. To isolate the performance of the aggregation operations from external factors such as file I/O and deserialization, the dataset is fully loaded into memory prior to execution. The queries used in this evaluation are as follows:

```
Low-Card: SELECT ss_promo_sk, ss_net_paid FROM
    store_sales ORDER BY ss_promo_sk;
High-Card: SELECT ss_ticket_number, ss_net_paid FROM
    store_sales ORDER BY ss_ticket_number;
Decimal-Sort: SELECT ss_sales_price, ss_quantity
    FROM store_sales ORDER BY ss_sales_price DESC;
Top-N: SELECT ss_net_profit, ss_ticket_number FROM
    store_sales ORDER BY ss_net_profit DESC LIMIT
    1000;
Multi-Column: SELECT ss_store_sk, ss_sold_date_sk,
    ss_net_paid FROM store_sales ORDER BY
    ss_store_sk ASC, ss_sold_date_sk DESC;
```

These queries test distinct sorting scenarios: 1) low-cardinality sort on ss_promo_sk (1,000 unique values, 0.0003% cardinality) versus high-cardinality sort on ss_ticket_number (24M unique values, 8.33% cardinality); 2) decimal sort on ss_sales_price in descending order, assessing non-integer numeric handling and comparison overhead; 3) multi-column sort on composite keys (ss_store_sk ascending, ss_sold_date_sk descending), evaluating CPU-intensive tuple comparison efficiency; and 4) Top-N sort with LIMIT 1000 on ss_net_profit, examining partial sort optimizations where engines aggregate top records per partition rather than performing full global sorting.

Figure 6 reports the performance comparison among different engines. Spark+ClickHouse consistently delivers the fastest execution times across all query types except Top N, achieving an average speedup of 2.50× over Vanilla Spark. Spark+Velox demonstrates moderate performance with an average speedup of 1.90×, while Spark+DataFusion shows the least improvement among the three engines with an average speedup of 1.39×.

Summary of Sort Performance: Spark+ClickHouse generally outperforms other engines across low-cardinality, high-cardinality, decimal, and multi-column sorting scenarios. Spark+Velox is competitive in the Top N query. Spark+DataFusion offers moderate performance.

3.7 Summary of Performance Comparison

We identify six dimensions in our benchmarking to compare the performance of the four evaluated Spark engines. A Kiviat diagram is used to summarize the performance across different workloads, shown in Figure 7. We employ a five-point scale to assess each engine's performance across various dimensions. For different workloads, systems are evaluated on these dimensions using a 0–5 scale, with 5 indicating the best performance. To assign the scores, we normalize each workload's runtimes to the range [0, 1], followed by log scaling. From Figure 7, we observe that:

(1) **Velox.** Velox outperforms or matches the performance of other query engines across all workloads, except for ORC table scan and sorting. In Section 5, we will present a case study to investigate the potential of hardware accelerators to address the performance gap in Velox's ORC table scan.

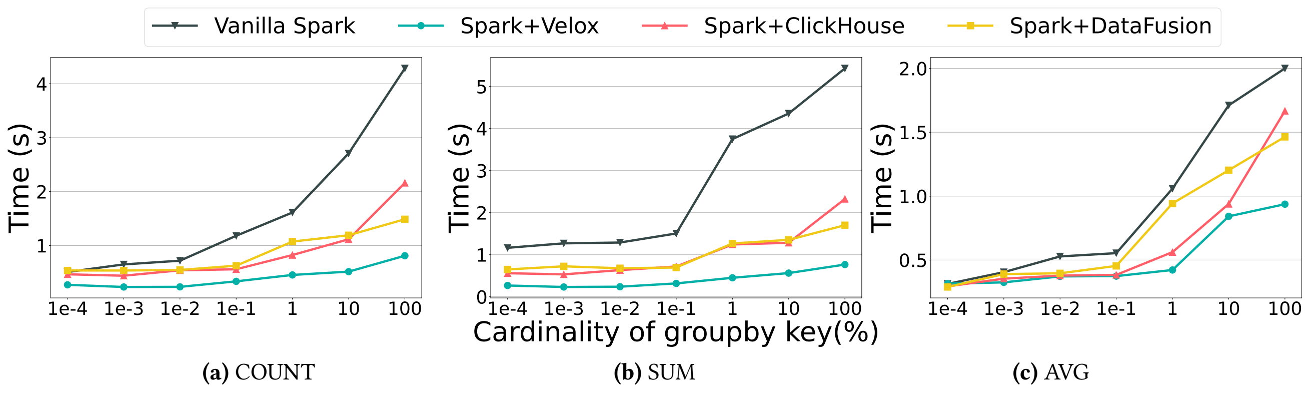


Figure 5: Group by hash aggregation performance with different cardinalities, under different engines

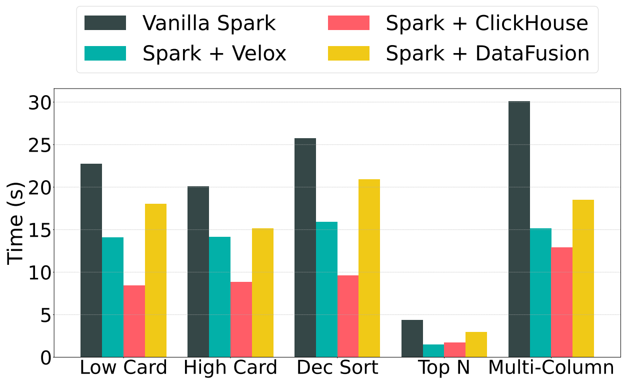


Figure 6: Sorting performance across different query variations under various engines.

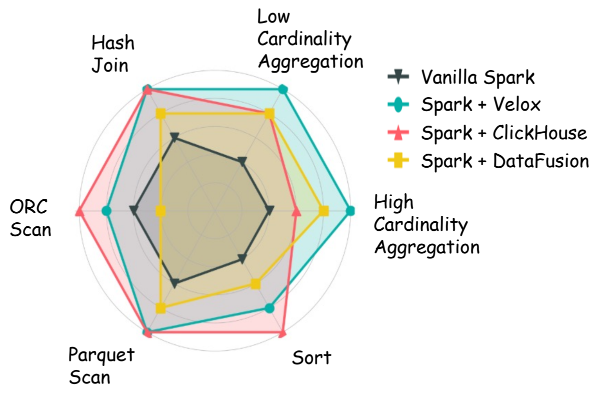


Figure 7: Performance characterization of the four engine under different analytics workload

(2) **ClickHouse.** ClickHouse excels in ORC table scan, sorting, Parquet table scan, and hash join, but underperforms in workloads involving aggregation operators, particularly in high-cardinality aggregation queries.
(3) **DataFusion.** DataFusion often lags behind, with no single category performing the best. It demonstrates some balanced performance across all workloads except ORC table scan.

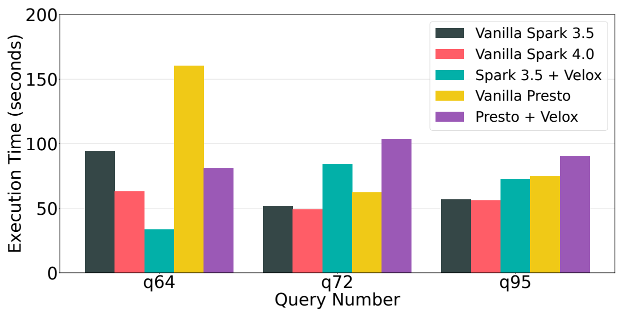


Figure 8: Join-heavy queries performance comparison of different Spark versions and Presto

4 Case Study 1: Code-Gen vs. Native Vectorized Execution in Join-Heavy Queries

In this case study, we first validate that our observation on performance degradation of native execution is universal by performing an ablation study. Then, we analyze the underlying causes of performance degradation in TPC-DS queries with multiple hash join operators when executed in native vectorized engines. Based on that, we propose a cost model based operator placement strategy to avoid the performance degradation and achieve further speedup.

4.1 Ablation Study of Big Data Engines

To validate that the observed performance degradation of native execution is not an artifact of a specific big data engine or version, we perform an ablation study focusing on join-heavy queries. First, to isolate the impact of Spark's evolution, we benchmark the baseline performance of Vanilla Spark 3.5 (3.5.5) against Spark 4.0. We then extend our analysis beyond the Spark ecosystem to generalize our findings. To this end, we also compare the JVM-based Presto engine (v0.283) [45] with its native execution, Prestissimo [37], which is accelerated by Velox. It should be noted that at the time of the experiments, Gluten-based Velox integration is not yet available for Spark 4.0. All cluster configurations for Spark and Presto are identical to those described in Section 3.

As shown in Figure 8, the results corroborate the performance patterns identified in Section 3.2. For Q64 query, native execution engines (Spark + Velox and Presto + Velox) consistently outperform their JVM-based code-gen counterparts. Conversely, for queries Q72 and Q95, we observe the opposite trend: the code-gen execution within the vanilla engines (Spark and Presto) yields better performance than their respective native versions. This consistent pattern across different engines (Spark vs. Presto) and versions (Spark 3.5 vs. 4.0) demonstrates that the performance trade-offs are inherent to the execution paradigms rather than implementation-specific artifacts. For the remainder of this paper, we continue to use Spark 3.5.5 for the experimental results.

4.2 In-depth Analysis of Join-Heavy Queries

Our analysis targets TPC-DS queries (Q64, Q72, and Q95) characterized by including more than 5 consecutive BroadcastHashJoin (BHJ) operators and where these join operators make up more than 40% of the total CPU time. These queries expose a fundamental performance trade-off between whole-stage code-generation (WSCG) and vectorized execution models. Spark's WSCG engine fuses consecutive non-pipeline-breaking joins into a single, tightly-looped function. Since BroadcastHashJoin's probing phase is non-pipeline-breaking, consecutive BHJs can be fused within a single nested loop, eliminating the need for intermediate vector materialization between join operations. This strategy processes intermediate tuples primarily in CPU registers, effectively eliminating inter-operator data materialization overhead. In contrast, native vectorized engines like Velox necessitates the materialization of columnar batches between every operator. While this materialization cost is conventionally considered negligible, our findings reveal it can become a bottleneck when the intermediate data footprint exceeds CPU cache capacity.

To investigate this trade-off, we conduct a micro-architectural analysis comparing vanilla Spark (WSCG) with Velox (vectorized). The results, summarized in Tables 4 and 5, reveal a direct link between intermediate data size and performance. For Q64, which generates a small intermediate data volume (with a max vector size of 0.03MB and an average of 0.02MB per partition), the vectorized engine excels. Its SIMD parallelism yields up to 2.7× fewer instructions, and the small data footprint ensures cache residency, making materialization costs negligible. Conversely, for Q72 and Q95, which generate large intermediate data, with average vector sizes of 2.7MB and 1.2MB per partition respectively, the vectorized engine suffers from severe cache pollution. This is evidenced by substantially higher cache miss rates (up to 3.9× L1 and 16.3× LLC misses) and lower instructions per cycle (IPC). The increased instruction count arises from additional memory operations for large intermediate vectors and branch instructions due to cache-miss handling. For these queries, WSCG's ability to bypass memory by keeping data in registers outweighs the computational benefits of vectorization, resulting in superior performance.

Table 4: Time and CPU counters of TPC-DS Q64, Q72 and Q95, with SF=1T, CPU counters values (instruction counts, CPU cycles, IPC, L1 miss and LLC miss) are normalized by number of tuples processed in each query. Spark denotes vanilla Spark in JVM.

Query	Engine	Total Time (s)	Join Time (s)	Instru -ctions	Cycles	IPC	L1 miss	LLC miss
Q64	Spark	**93.9**	**41.4**	**48**	**58**	0.83	0.16	0.03
Q64	Velox	31.9	16.7	18	20	**0.88**	0.16	0.02
Q72	Spark	51.7	31.0	174	129	**1.35**	1.81	0.11
Q72	Velox	**84.0**	**77.3**	**248**	**210**	1.18	**6.31**	**1.14**
Q95	Spark	56.8	28.3	160	142	**1.13**	1.42	0.06
Q95	Velox	**72.5**	**59.2**	**173**	**181**	0.96	**5.55**	**0.98**

Table 5: Intermediate vector size (MB) per CPU core between join operators of TPC-DS Q64, Q72 and Q95 at SF=1TB

Query	Max Vec Size	Avg Vec Size	Max Total Size	Avg Total Size	Max Vec Num	Avg Vec Num
Q64	0.03	0.02	60.1	11.1	2,003	555
Q72	4.6	2.7	6,041.6	2,785.3	1,313	1,032
Q95	1.3	1.2	1,500.2	980.4	1,154	817

These findings highlight a critical insight: the mandatory materialization in native vectorized engines, while efficient for cache-resident data, introduces a severe memory bottleneck for workloads with large intermediate vector size. The eviction of these large vectors from CPU caches to main memory negates the performance gains from vectorized execution. A more efficient and intuitive approach would employ a hybrid execution model, which can selectively revert to code-generated operator execution for query plan projected to produce large, cache-unfriendly intermediate data, thus strategically bypassing the materialization penalty.

4.3 Cost Based Operator Placement

To validate our insights and enhance the performance within Spark's native execution system, we propose a cost model guided operator placement strategy. As shown in Figure 9, after the Catalyst optimizer generates a physical plan, our method selects between the vectorized and whole-stage code-gen engines for optimal operator execution. A key contribution of our work is a cost model that accounts for the inter-operator vector materialization overhead inherent to the vectorized engine. This overhead, which arises from materializing vector batches between each operator, is distinct from the cost of pipeline-breaking operations (e.g., sort) that affect both engines. The code-gen engine mitigates this specific overhead by fusing operators into a nested loop in a single function where data is kept within registers. Our model extends previous work on individual hash-joins [22, 23] by analyzing the BroadcastHashJoin pipeline and incorporating costs of distributed execution and data format conversions during engine transition. We parameterize the model using Spark's online cardinality estimates and offline sampling to select the plan with the minimum overall execution cost. In the following, we introduce the formulation of our proposed cost model, the notations are listed in Table 6 for ease of reference.

4.3.1 Modeling of Individual Hash Join. **In the build phase** of the hash join, all the tuples in the build side table B are scanned from main memory, and a hash function is applied to each tuple's join key to build an in-memory hash table. The hash table is then written back to main memory. The memory cost includes reading of the build side table and writing the hash table, while the compute cost reflects the hashing cost.

$$C_{\text{buildMem}} = w_{\text{buildMem}} \cdot p \cdot \left(\frac{|B|}{BW_{\text{mem\_r}}} + \frac{|H|}{BW_{\text{mem\_w}}}\right) \tag{1}$$

$$C_{\text{buildCompute}} = w_{\text{buildCompute}} \cdot p \cdot \frac{|B|}{BW_{\text{hash}}} \tag{2}$$

In the probe phase, the cost includes loading the probe-side table from main memory and probing the hash table by applying the hash function to find matching tuples. Every probe access requires reading an entire cache line. Because we are constructing

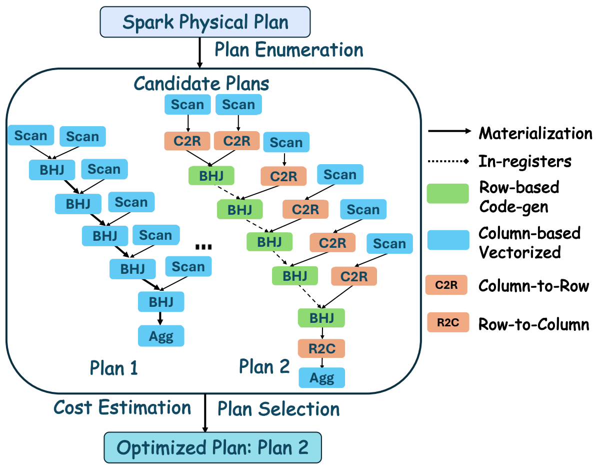


Figure 9: Workflow of cost model guided operator placement

Table 6: Notation of cost model

Notation	Description
$C_{..}$	Cost.
$\|B\|$	Size of the build table.
$\|H\|$	Size of the in-memory hash table.
$\|P\|$	Size of the probe table.
$\|I_{..}\|$	Size of intermediate vector between operators.
$Card_P$	Cardinality of the probe table.
p	Parallelism (number of partitions).
$w_{..}$	Model coefficients in each stage.
CL	Cacheline size.
CM	Last level cache miss rate.
$BW_{\text{mem\_w}}$	Write bandwidth of the memory device.
$BW_{\text{mem\_r}}$	Read bandwidth of the memory device.
BW_{network}	Bandwidth of the network.
BW_{hash}	Hashing operation processing rate.
BW_{ser}	Serialization processing rate of C2R operation.
BW_{deser}	De-serialization processing rate of R2C operation.

a cost model for large-scale data (1TB) and the hash table is assumed to be much larger than the last-level cache (L3 cache), we assume every cache miss results in a main memory access.

$$C_{\text{probeMem}} = w_{\text{probeMem}} \cdot p \cdot \left(\frac{|P|}{BW_{\text{mem\_r}}} + CM \cdot \frac{Card_P \cdot CL}{BW_{\text{mem\_r}}} \right) \quad (3)$$

$$C_{\text{probeCompute}} = w_{\text{probeCompute}} \cdot p \cdot \frac{|P|}{BW_{\text{hash}}} \quad (4)$$

Assuming that hash join is either compute-bound or memory-bandwidth-bound, the total cost of the hash join build and probe stage is defined as follows:

$$C_{\text{hashJoinLocal}} = \max \begin{cases} C_{\text{buildCompute}} + C_{\text{probeCompute}} \\ C_{\text{buildMem}} + C_{\text{probeMem}} \end{cases} \quad (5)$$

In the data exchange phase of the BroadcastHashJoin, each of the p tasks receives the build table $|B|$ from other tasks, except for the $1/p$ portion it initially holds. The total network transfer volume is approximately $(p-1) \cdot |B|$. The total cost of the p tasks can be modeled as:

$$C_{\text{broadcast}} = w_{\text{broadcast}} \cdot (p-1) \cdot \frac{|B|}{BW_{\text{network}}} \quad (6)$$

Finally, the total cost of each join type can be expressed as:

$$C_{\text{broadcastHashJoin}} = C_{\text{broadcast}} + C_{\text{hashJoinLocal}} \quad (7)$$

4.3.2 Modeling of R2C and C2R Overhead. Falling back from columnar native execution to row-based code-gen execution necessitates Column-to-Row (C2R) conversion, while offloading from row-based to columnar based execution also involves Row-to-Column (R2C) conversion. We model these conversion costs based on the widely used native engine Velox. R2C involves deserializing Spark's `UnsafeRow` to a Velox vector, and C2R involves serializing a Velox vector to Spark's `UnsafeRow`. The conversion costs are defined as:

$$C_{\text{r2c}} = w_{\text{r2c}} \cdot \frac{|I_{\text{row}}|}{BW_{\text{deser}}} \qquad C_{\text{c2r}} = w_{\text{c2r}} \cdot \frac{|I_{\text{col}}|}{BW_{\text{ser}}} \quad (8)$$

4.3.3 Modeling of Data Movement of Materialization in Vectorized Engine. In this section, we model the overhead incurred by vector materialization across the cache-memory boundary. This scenario arises when the vector batch size is large enough to exceed the capacity of the CPU cache, thereby triggering cache spilling. The model rests on three assumptions: 1) available memory is sufficient to prevent memory spilling to disk, 2) data movement follows a sequential access pattern, and 3) vectorized execution with large intermediate data leads to a memory-bandwidth-bound scenario. Since materializing a vector batch requires writing intermediate results to memory and then reading them back, the total cost combines both write and read overheads:

$$C_{\text{mater}} = w_{\text{w}} \cdot \frac{|I|}{BW_{\text{mem\_w}}} + w_{\text{r}} \cdot \frac{|I|}{BW_{\text{mem\_r}}} \quad (9)$$

where w_{w} and w_{r} are weighting factors that capture the efficiency of memory access during the write and read phases, respectively.

4.3.4 Modeling of Pipeline with Consecutive BroadcastHashJoin. The probe phase of BroadcastHashJoin preserves pipeline continuity, eliminating the intermediate result materialization typically required at pipeline boundaries. However, vectorized execution engines still incur the inherent overhead of vector batch materialization dictated by their interpreted model, a cost that becomes especially pronounced when intermediate vectors exceed cache capacity, forcing data movement across the cache-memory boundary. The total cost of executing N vectorized columnar operators is modeled as:

$$C_{\text{vectorized}} = \sum_{i=1}^{N} w_{\text{vectorized}} \cdot C^{i}_{\text{hashJoin}} + \sum_{i=1}^{N} C^{i}_{\text{mater}} \quad (10)$$

where $\sum_{i=1}^{N} C^{i}_{\text{mater}}$ denotes the cumulative cost of materializing intermediate results during vectorized execution.

By contrast, in query pipelines with code-gen operators, the intermediate vector batch materialization overhead is eliminated. However, falling back from the columnar-based vectorized execution to row-based code-generated execution introduces additional overhead from data format conversions between columnar and row representations. The cost model in this case is given by:

$$C_{\text{codegen}} = \sum_{i=1}^{N} w_{\text{codegen}} \cdot C^{i}_{\text{hashJoin}} + \sum_{i=1}^{T} C^{i}_{\text{c2r}} + C_{\text{r2c}} \quad (11)$$

where $\sum_{i=1}^{T} C^{i}_{\text{c2r}}$ and C_{r2c} represent the overhead of columnar-to-row and row-to-column format conversions, respectively, and T represents the number of build and probe tables involved in the code-gen hash join pipeline.

4.4 Results of Cost Guided Operator Placement

We evaluate our cost model guided operator placement strategy on nine TPC-DS queries, each containing at least five BroadcastHashJoin operators. Figure 10 shows execution times with three configurations: our model (red), Velox vectorized execution (blue), and JVM-based code-gen execution (cyan). The total execution time is broken down into contributions from vectorized hash join, code-gen hash join, row/column conversion, and other operators.

For *joins* in Q13, Q29, Q72, Q85, and Q95, Velox underperform Spark's code-gen pipeline due to vector batch materialization overhead exceeding native execution benefits; note Velox still performs better for other operators. Our model therefore selects code-gen mode for the joins while offloading tasks like table scans to native execution, incurring only minimal conversion overhead. This achieves 1.50× average speedup (up to 2.12× speedup) over Velox for these five queries, while the remaining four show negligible change. Against vanilla Spark, our approach demonstrates speedup for all queries, with an average speedup of 1.81× and a maximum speedup of 2.96×.

5 Case Study 2: Heterogeneous Acceleration on Apache ORC Table Scan

With the significant slowdown of Moore's law, hardware accelerators are becoming more important for enhancing performance in compute-intensive workloads. Extensive research has investigated the acceleration of analytic workloads using heterogeneous accelerators. For example, many studies have explored the use of GPUs to optimize performance-critical database operations such as hash joins [26, 27, 41], aggregations [28], decompression [32], and end-to-end query execution [6, 55]. Similarly, customizable accelerators like FPGAs have been extensively studied for parallel computation and data stream processing tasks, including data (de)compression [17], decoding [51], filtering [24], and shuffle operations [57].

Besides accelerating individual operators, integrating heterogeneous hardware into JVM-based compute engines, such as Apache Spark, has also gained significant attention [42, 50, 54]. However, limited support for low-level hardware control in the JVM and the high cost of data communication between the JVM and hardware often hinder performance [5, 36]. Offloading workloads to native execution engines written in C++ or Rust offers a potential solution to these challenges. Native languages provide better support for device-level programming frameworks (e.g., CUDA, OpenCL) and can significantly reduce data transfer overhead by bypassing the JVM. This approach can unlock the full computational potential of hardware accelerators in big data systems such as Spark.

While hardware acceleration combined with native execution looks promising, we need to address a few critical questions:

(1) **Operator Selection**: Which operators remain as bottlenecks after offloading to native execution, and which specific operators should be offloaded to hardware accelerators?
(2) **Hardware Suitability**: Which type of hardware is best suited for accelerating big data analytics workloads?
(3) **Performance Gains**: What is the potential performance benefit of utilizing heterogeneous accelerators after native execution?

5.1 Bottlenecks after Native Execution

Before exploring the offloading of computational workloads to hardware accelerators, it is essential to quantitatively analyze the bottlenecks and identify which operators are most suitable for offloading after native execution. Figure 11 presents the CPU execution time breakdown for the TPC-DS queries at a scale factor of 1T, utilizing Velox as the native engine within Spark and Apache ORC as the file format. The result is profiled on the same Spark cluster as section 3 using *async-profiler*. Our analysis reveals that ORC table scan dominates among all operators, accounting for 44.8% runtime.

Within the table scan workload, decompression and decoding operations are the most time-consuming, comprising 23.1% and 16.6% of the total runtime, respectively, whereas I/O operations constitute only 5.1%. Since I/O operations are inherently asynchronous and can often be overlapped with computation, this profiled value represents only the I/O portion that could not be hidden by the CPU's other work, not the total I/O time. Furthermore, as depicted in Figure 7, Velox exhibits superior performance compared to alternative native engines across most analytical workloads, with the notable exception of ORC table scan operations. These empirical observations inform our decision to target the ORC table scan operator as the primary candidate for hardware acceleration. Our case study endeavors to address this specific performance bottleneck within Velox and to systematically assess the potential benefits of various hardware accelerators in enhancing ORC table scan performance.

5.2 Hardware Acceleration with ORC Table Scan

With the advent of high-bandwidth SSDs, storage I/O overhead has been substantially mitigated. As a result, the performance bottleneck of table scanning operations has shifted to CPU-bound tasks, where decoding and decompression operations alone can account for more than 90% of the total CPU time. To systematically evaluate the performance of hardware acceleration for table scan acceleration, we conduct a comprehensive case study using a single integer-column ORC file with multiple execution engines in diverse hardware platforms. Our testbed includes:

(1) **CPU-based implementations:** Apache Spark + Velox engine and Apache Spark + ClickHouse. The software versions are consistent with those described in Section 3.1.
(2) **GPU-based implementations:** Nvidia RAPIDS Spark accelerator (v24.08.1) on an Nvidia V100 GPU, which utilizes PCIe 3.0 as the interconnect between the CPU and GPU for high-bandwidth data transfer.
(3) **FPGA-based implementations:** FPGA-accelerated ORC table scan called FORC [51] on an AMD/Xilinx Alveo U280 FPGA, with PCIe 3.0 serving as the interconnect between the CPU and FPGA. Note in our evaluated FORC version, both ORC decompression and decoding have been accelerated on the FPGA.

Figure 12(a) presents the throughput comparison of ORC table scans across different engines and accelerators. Both GPU- and FPGA-based solutions deliver higher throughput than CPU-native engines. Specifically, the GPU-based RAPIDS cuDF implementation achieves a geomean speedup of 2.94× over Velox, while the FPGA-based FORC implementation attains a geomean speedup of 7.69× over Velox.

The FPGA-based FORC accelerator delivers superior throughput, achieving a 2.43× geomean speedup over GPU-based RAPIDS

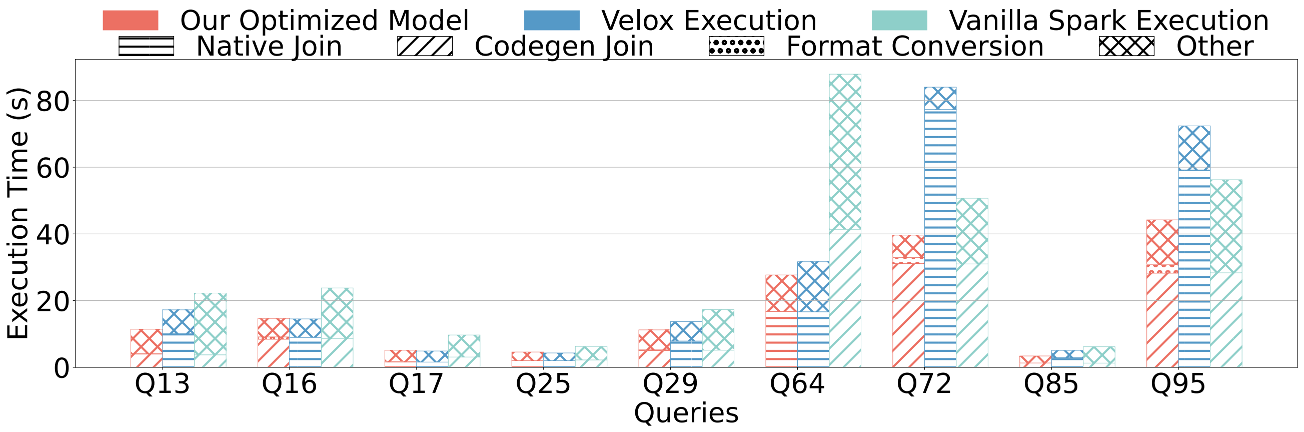


Figure 10: Performance comparison of the TPC-DS (SF=1T) queries with at least five consecutive BroadcastHashJoin with our cost guided optimized plan, Velox native execution plan, and vanilla Spark code-gen plan

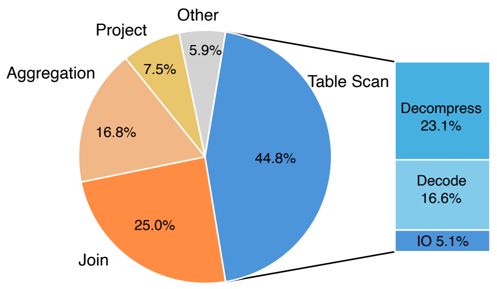


Figure 11: CPU time breakdown for TPC-DS queries with SF=1T, with Velox native execution and ORC file format

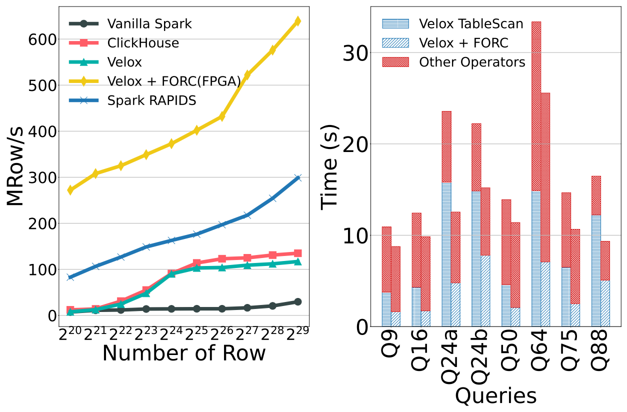


Figure 12: (a) Comparative analysis of ORC table scan throughput across different query engines and accelerators. (b) End-to-end performance comparison for scan-intensive TPC-DS queries (SF=1T): Velox versus Velox + FORC.

cuDF for ORC table scan. This performance advantage stems from two architectural strengths. First, the reconfigurable nature of FPGAs enables tailored optimizations for ORC's hybrid compression and encoding schemes, such as run-length encoding. Second, the FORC accelerator employs a finer-grained dataflow architecture that seamlessly overlaps decompression, decoding, and filtering stages while integrating I/O operations and CPU-FPGA data transfers directly into the streaming pipeline. These architectural distinctions position the FORC accelerator as a compelling solution for high-performance ORC data processing in large-scale analytics.

We further integrate FORC with Velox ORC reader and evaluate end-to-end performance using TPC-DS table scan heavy queries (Q9, Q16, Q24a, Q24b, Q50, Q64, Q75, and Q88); note Q64 is both join heavy and scan heavy. Figure 12(b) presents the breakdown of the total query execution time and table scan time. With the FORC accelerator, we observe up to a 3.3× speedup in the table scan operator and up to a 1.8× improvement in the overall query execution time.

6 Related Work

Benchmarking of Big Data Systems. Wu et al. [53] introduce Raven to evaluate cloud OLAP engines (e.g., Presto, Spark SQL) across diverse configurations. Theodorakopoulos et al. [46] benchmark Spark GraphX and Flink for graph tasks, while DIAMetrics [11] offers a modular framework for query engine monitoring. However, these studies focus on JVM-based engines, neglecting the native execution engines addressed in this paper.

Query Optimization Unifying Code Generation and Vectorized Execution. Some vectorized engines [39] support code generation for specific tasks, while compiled engines [29, 52] partially adopt vectorization. Yet, none utilizes a cost model for operator placement. The exception, Tupleware [10], uses a cost model to select between execution strategies but is limited to UDFs. In contrast, our model targets hash joins and is extensible to all operators.

Heterogeneous Acceleration for Big Data Engines. Xekalaki et al. [54] introduce TornadoVM [13] for heterogeneous JIT compilation and scheduling. However, unlike our C++ native integration, it suffers from costly JVM-hardware data transfers. Lu et al. [25] propose SQL2FPGA to translate Spark SQL to FPGA programs but focus limitedly on hash join/aggregation without addressing table scans or comparing accelerator performance.

7 Discussion

We also combine our two optimizations, cost model guided hybrid execution (Opt1) presented in Section 4 and FPGA-accelerated table scan (Opt2) presented in Section 5, and evaluate the performance for the top 20 TPC-DS queries in Figure 13. On average, the cost model guided hybrid execution (Opt1) achieves a 1.14×

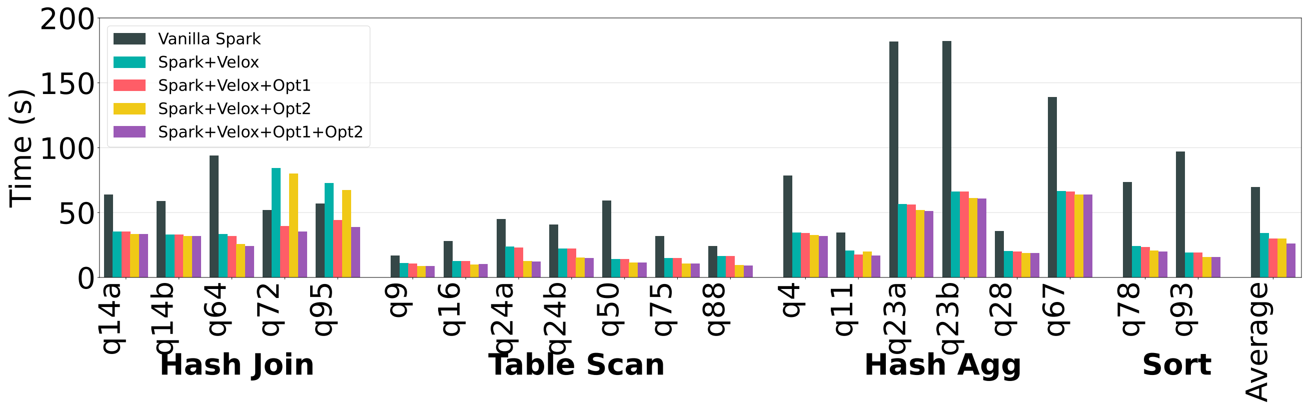


Figure 13: Execution time comparison of Top 20 TPC-DS queries showing the performance impact of cost model guided hybrid execution (Opt1), FPGA-accelerated table scan (Opt2), and the combined optimizations (Opt1+Opt2). Queries are grouped by the query bottleneck.

speedup (up to 2.1× speedup) over the baseline Spark+Velox configuration, while the FPGA-accelerated table scan (Opt2) delivers a 1.25× speedup (up to 1.8× speedup). For join-heavy queries like Q72 and Q95, Opt2 underperforms Vanilla Spark because its acceleration of table scans is outweighed by the slower performance of Velox's vectorized join compared to Spark's code generation; for these two queries, Opt1 achieves a significant speedup. When both optimizations are combined (Opt1+Opt2), the system achieves a 1.46× average speedup (up to 2.32× speedup) over Spark+Velox. These performance gains highlight two key architectural directions that deserve further investigation: hybrid execution models and unified heterogeneous hardware abstraction.

Engines with Hybrid Execution Models: While we retain some operators in code-gen execution to avoid materialization overhead, Spark's code-gen remains limited by JVM constraints. Future work could explore native code generation in C++ or Rust for better performance. The performance degradation from vector materialization suggests the need for unified solutions that combine code-gen execution with vectorized execution, similar to systems like InkFuse [52]. Additionally, cost models should extend beyond join operators to support a broader range of operators. As hardware accelerators expand the query plan space, unified cost models become crucial for choosing between vectorized engines, code-gen engines, and various accelerator designs.

Unified Heterogeneous Hardware Abstraction: The FPGA table scan case study, achieving up to 3.3× speedup for table scan itself and up to 1.8× speedup for end-to-end queries, highlights the potential of heterogeneous hardware acceleration to address performance gaps in native engines. Frameworks that dynamically allocate workloads across CPUs, GPUs, and FPGAs are essential. Native engines like Velox enable accelerator integration through operator specialization [35]. While Velox-cuDF [48] and Velox Wave [49] propose GPU-focused interfaces, future research needs to tackle two challenges: 1) developing methods for dynamically mapping operators to the optimal hardware acceleration platform (beyond GPU) based on cost-performance characteristics [4], and 2) enabling efficient data movement across platforms via unified in-memory data formats.

8 Conclusion

In this work, we provide in-depth comparative performance analysis and insights for existing Spark native execution engines and demonstrate that offloading workloads to native execution does not universally yield performance improvements due to materialization overhead of vectorized execution. Based on these insights, we propose a cost model driven operator placement strategy that intelligently selects the implementation of the query operator between vectorized and code-gen execution, achieving no performance degradation and up to 2.1× speedup compared to pure native execution. Furthermore, we identify the remaining bottlenecks in native execution and leverage the native engine as a unified interface to offload decode and decompression workload in ORC table scan to heterogeneous hardware accelerators, achieving performance improvements of up to 1.8× in end-to-end TPC-DS query.

9 Artifacts

The code and software that is developed for this work and used for all the presented experiments is made publicly available at https://github.com/SFU-HiAccel/NativeEngineBenchmark.

Acknowledgments

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

References

[1] Apache Arrow. 2025. Apache Arrow. https://arrow.apache.org/.
[2] Auron. 2025. Auron. https://github.com/apache/auron.
[3] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.
[4] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. 2014. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1609–1612.
[5] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 29–29.
[6] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment* 12, 5 (2019), 544–556.

[7] Apache DataFusion Comet. 2025. Apache DataFusion Comet. https://datafusion.apache.org/comet/.

[8] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[9] Jason Cong, Peng Wei, and Cody Hao Yu. 2018. From JVM to FPGA: bridging abstraction hierarchy via optimized deep pipelining. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*. 21.

[10] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling UDF-centric workflows. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1466–1477.

[11] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2021. Diametrics: benchmarking query engines at scale. *ACM SIGMOD Record* 50, 1 (2021), 24–31.

[12] Apache Flink. 2025. Apache Flink. https://flink.apache.org/.

[13] Juan Fumero, Michail Papadimitriou, Foivos S Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 165–178.

[14] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. 2008. Parallel computing experiences with CUDA. *IEEE micro* 28, 4 (2008), 13–27.

[15] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[16] Max Grossman, Shams Imam, and Vivek Sarkar. 2015. Hj-opencl: Reducing the gap between the jvm and accelerators. In *Proceedings of the principles and practices of programming on the java platform*. 2–15.

[17] Tobias Hahn, Stefan Wildermann, and Jürgen Teich. 2024. JSON-CooP: A JSON Decompression/Parsing Co-Design for FPGAs. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 11–18.

[18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222.

[19] Giorgi Kikolashvili. 2019. *On the design of a JVM-based vectorized Spark query engine.* Ph. D. Dissertation. Universiteit van Amsterdam.

[20] 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*. 5–17.

[21] Jiang Li, Qi Xie, Yan Ma, Jian Ma, Kunshang Ji, Yizhong Zhang, Chaojun Zhang, Yixiu Chen, Gangsheng Wu, Jie Zhang, et al. 2023. Big Data Analytic Toolkit: A General-Purpose, Modular, and Heterogeneous Acceleration Toolkit for Data Analytical Engines. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3702–3714.

[22] Feng Liang, Francis CM Lau, Heming Cui, Yupeng Li, Bing Lin, Chengming Li, and Xiping Hu. 2024. RelJoin: Relative-cost-based selection of distributed join methods for query plan optimization. *Information Sciences* 658 (2024), 120022.

[23] Feilong Liu and Spyros Blanas. 2015. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 153–166.

[24] Kenneth Liu, Alec Lu, and Zhenman Fang. 2024. BitBlender: Scalable Bloom Filter Acceleration on FPGAs with Dynamic Scheduling. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 325–331. doi:10.1109/FPL64840.2024.00052

[25] Alec Lu, Jahanvi Narendra Agrawal, and Zhenman Fang. 2024. SQL2FPGA: Automated Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms. *ACM Transactions on Reconfigurable Technology and Systems* 17, 3 (2024), 1–28.

[26] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1633–1649.

[27] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *Proceedings of the 2022 International Conference on Management of Data*. 1017–1032.

[28] Gabriele Mencagli, Patrizio Dazzi, and Massimo Coppola. 2024. Springald: GPU-Accelerated Window-Based Aggregates Over Out-of-Order Data Streams. *IEEE Transactions on Parallel and Distributed Systems* 35, 9 (2024), 1657–1671.

[29] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[30] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[31] Apache ORC. 2025. Apache ORC. https://orc.apache.org/.

[32] Jeongmin Park, Zaid Qureshi, Vikram Mailthody, Andrew Gacek, Shunfan Shao, Mohammad AlMasri, Isaac Gelado, Jinjun Xiong, Chris Newburn, I-hsin Chung, et al. 2023. CODAG: Characterizing and Optimizing Decompression Algorithms for GPUs. *arXiv preprint arXiv:2307.03760* (2023).

[33] Apache Parquet. 2025. Apache Parquet. https://parquet.apache.org/.

[34] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proceedings of the VLDB Endowment* 15, 12 (Aug. 2022), 13.

[35] 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. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685.

[36] Johan Peltenburg, Ahmad Hesam, and Zaid Al-Ars. 2017. Pushing big data into accelerators: Can the JVM saturate our hardware?. In *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers 32*. Springer, 220–236.

[37] Prestissimo. 2025. Prestissimo. https://prestodb.io/docs/0.285/develop/presto-native.html.

[38] Protobuf. 2025. Protobuf. https://github.com/protocolbuffers/protobuf.

[39] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse-Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[40] 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 *VLDB Workshops*.

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

[42] Weiren Song, Jia You, and Xisheng Li. 2022. Accelerated Spark Streaming based on FPGA. In *2022 China Automation Congress (CAC)*. 1594–1599.

[43] Apache Spark. 2025. Apache Spark. https://spark.apache.org/.

[44] Substrait. 2025. Substrait. https://substrait.io/.

[45] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A Decade of SQL Analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[46] Leonidas Theodorakopoulos, Aristeidis Karras, Alexandra Theodoropoulou, and Georgios Kampiotis. 2024. Benchmarking Big Data Systems: Performance and Decision-Making Implications in Emerging Technologies. *Technologies* 12, 11 (2024), 217.

[47] TPC-DS. 2025. TPC-DS. https://www.tpc.org/tpcds/.

[48] Velox. 2025. Velox cudf. https://github.com/facebookincubator/velox/tree/main/velox/experimental/cudf.

[49] Velox. 2025. Velox Wave. https://github.com/facebookincubator/velox/tree/main/velox/experimental/wave.

[50] Tudor Alexandru Voicu and Zaid Al-Ars. 2019. SparkJNI: A Toolchain for Hardware Accelerated Big Data Apache Spark. In *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*. 152–157.

[51] Abdul Wadood, Alec Lu, Ken Zhang, and Zhenman Fang. 2024. FORC: A High-Throughput Streaming FPGA Accelerator for Optimized Row Columnar File Decoders in Big Data Engines. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 318–324.

[52] Benjamin Wagner, Andre Kohn, Peter Boncz, and Viktor Leis. 2024. Incremental Fusion: Unifying Compiled and Vectorized Query Execution . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 462–474.

[53] Tongyu Wu, Rong Gu, Yang Li, Hongbin Ma, Yi Chen, Ying Zhu, Xiaoxiang Yu, Tengting Xu, and Yihua Huang. 2023. Raven: Benchmarking Monetary Expense and Query Efficiency of OLAP Engines on the Cloud. In *International Conference on Database Systems for Advanced Applications*. Springer, 593–605.

[54] Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos Kotselidis. 2022. Enabling transparent acceleration of big data frameworks using heterogeneous hardware. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3869–3882.

[55] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2491–2503.

[56] Bowen Yu, Guanyu Feng, Huanqi Cao, Xiaohan Li, Zhenbo Sun, Haojie Wang, Xiaowei Zhu, Weimin Zheng, and Wenguang Chen. 2021. Chukonu: a fully-featured high-performance big data framework that integrates a native compute engine into Spark. *Proceedings of the VLDB Endowment* 15, 4 (2021), 872–885.

[57] Chen Zou, Hui Zhang, Andrew A. Chien, and Yang Seok Ki. 2021. PSACS: Highly-Parallel Shuffle Accelerator on Computational Storage. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 480–487.