Photon: A Fast Query Engine for Lakehouse Systems

Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, Matei Zaharia
photon-paper-authors@databricks.com
Databricks Inc.

ABSTRACT

Many organizations are shifting to a data management paradigm called the "Lakehouse," which implements the functionality of structured data warehouses on top of unstructured data lakes. This presents new challenges for query execution engines. The execution engine needs to provide good performance on the raw uncurated datasets that are ubiquitous in data lakes, and excellent performance on structured data stored in popular columnar file formats like Apache Parquet. Toward these goals, we present Photon, a vectorized query engine for Lakehouse environments that we developed at Databricks. Photon can outperform existing cloud data warehouses in SQL workloads, but implements a more general execution framework that enables efficient processing of raw data and also enables Photon to support the Apache Spark API. We discuss the design choices we made in Photon (e.g., vectorization vs. code generation) and describe its integration with our existing SQL and Apache Spark runtimes, its task model, and its memory manager. Photon has accelerated some customer workloads by over 10× and has recently allowed Databricks to set a new audited performance record for the official 100TB TPC-DS benchmark.

ACM Reference Format:
Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, Matei Zaharia . 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526054

1 INTRODUCTION

Enterprises today store a vast majority of their data in scalable, elastic *data lakes* such as Amazon S3, Azure Data Lake Storage, and Google Cloud Storage. These data lakes hold raw, often uncurated datasets in open file formats such as Apache Parquet or Delta Lake [4, 18], and can be accessed with a variety of engines, such as Apache Spark and Presto [49, 58], to run workloads ranging from SQL to machine learning. Traditionally, for the most demanding SQL workloads, enterprises have also moved a curated subset of their data into data warehouses to get high performance, governance and concurrency. However, this two-tier architecture is complex and expensive, as only a subset of data is available in the warehouse, and this data may be out of sync with the raw data due to issues in the extract, transform and load (ETL) process [19].

In response, many organizations are shifting to a data management architecture called the *Lakehouse* [19], which implements data warehouse features such as governance, ACID transactions and rich SQL support directly over a data lake. This single-tier approach promises to simplify data management, as users can govern and query *all* their data in a uniform way, and there are fewer ETL steps and query engines to manage. Recently, new storage layers such as Delta Lake [18] have enabled many of the management features of data warehouses—such as transactions and time travel—on data lakes, and have provided useful tools for optimizing storage access, such as data clustering and data skipping indices. However, maximizing the performance of Lakehouse workloads requires optimizing not only the storage layer, but also query processing.

This paper presents Photon, a new vectorized query engine we developed at Databricks for Lakehouse workloads that can execute queries written in either SQL or in Apache Spark's DataFrame API [20]. Photon has already executed tens of millions of queries from hundreds of customers. With Photon, our customers have observed average speedups of 3× over our previous Databricks Runtime (an optimized engine based on Apache Spark), and maximum speedups of over 10×. Databricks also set an audited 100TB TPC-DS world record in November 2021 with Photon on a Lakehouse system using the Delta Lake format on Amazon S3, showing that state-of-the-art SQL performance is attainable with open data formats and commodity cloud storage.

Designing Photon required tackling two key challenges. First, unlike a traditional data warehouse engine, Photon needed to perform well on *raw, uncurated data*, which can include highly irregular datasets, poor physical layout, and large fields, all with no useful clustering or data statistics. Second, we wanted Photon to support, and be semantically compatible with, the existing Apache Spark DataFrame API that is widely used for data lake workloads. This was critical to deliver the Lakehouse promise of a *single* query engine with uniform semantics for all of an organization's workloads, but created difficult engineering and testing challenges. Of course, subject to these two challenges, we wanted Photon to be as fast as possible. We describe how these two challenges led us to Photon's design: a vectorized engine written in C++, that interfaces cleanly with Apache Spark's memory manager and that includes a variety of optimizations for raw, uncurated data.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526054

Challenge 1: Supporting raw, uncurated data. The Lakehouse environment challenges query engines with a greater variety of data than traditional SQL warehouses. On one end of the spectrum, for the "clean" tabular datasets in the Lakehouse, users take great care to clean and organize their data for read performance by designing schemas with constraints, statistics, and indices. On the other end of the spectrum, uncurated raw data may have sub-optimal data layouts like small files, many columns, sparse or large data values, and no useful clustering or statistics—but many users also wish to query such data. In addition, strings are convenient and prevalent in raw data, even to represent numeric data like integers and dates. This data is also frequently denormalized, so string columns may additionally use placeholder values for unknown or missing values instead of NULL, and schema information such as nullability or string encoding (e.g., ASCII vs. UTF-8) is typically absent.

As a result, an execution engine for the Lakehouse must have a design that is flexible enough to deliver good performance on arbitrary uncurated data, and excellent performance on data following Lakehouse best practices—multidimensional clustering [19], reasonable file sizes, and appropriate data types, for example—across a variety of use cases such as data science, ETL, ad hoc SQL, and BI.

We addressed this challenge with two early design decisions. First, we chose to build the engine using the *vectorized-interpreted model in lieu of code generation*, unlike Spark SQL's choice to use code generation [20]. Vectorized execution enabled us to support *runtime adaptivity*, wherein Photon discovers, maintains, and exploits micro-batch data characteristics with specialized code paths to adapt to the properties of Lakehouse datasets for optimal performance. For example, Photon runs optimized per-batch code for columns that rarely have NULL values, or mostly-ASCII string data.

We also observed other engineering advantages with the vectorized approach. Although we found some scenarios where the code generation model delivers better performance (e.g., complex conditional expressions), our experience while prototyping both approaches and from working on other engines was that the vectorized model was easier to build, profile, debug, and operate at scale. This allowed us to invest more time in specializations that narrowed the performance gap between the two. Preserving abstraction boundaries such as query operators also facilitates collecting rich metrics to help end users better understand query behavior.

Second, we chose to *implement Photon in a native language* (C++) rather than following the existing Databricks Runtime engine, which used the Java VM. One reason for this decision was that we were hitting performance ceilings with the existing JVM-based engine. Another reason for switching to native code was internal just-in-time compiler limits (e.g., on method size) that created performance cliffs when JVM optimizations bailed out. Finally, we found that the performance of native code was generally easier to explain than the performance of the JVM engine, since aspects like memory management and SIMD were under explicit control. The native engine not only delivered a performance boost, but also allowed us to handle large record sizes and query plans more easily.

Challenge 2: Supporting existing Spark APIs. Organizations already run a variety of applications over their data lakes, ranging from ETL workloads to advanced analytics such as machine learning. On the Databricks platform, these workloads leverage Apache Spark's APIs, using a mix of DataFrame or SQL code that goes through a SQL optimizer and user-defined code that is treated as a black box. To accelerate these existing workloads and ensure that SQL workloads on Databricks experience the same semantics as Spark workloads, we designed Photon to integrate with the Spark engine and to support Spark workloads (with a mix of SQL operators and UDFs) as well as pure SQL workloads. This was challenging because Photon had to be able to share resources with user-defined code (as happens in Spark), and had to match the semantics of Apache Spark's existing Java-based SQL engine.

To address this challenge, Photon integrates closely with the Apache Spark-based Databricks Runtime (DBR). DBR is a fork of Apache Spark that provides the same APIs, but contains improvements to reliability and performance. Photon fits into DBR as a new set of physical operators that can be used for part of the query plan by DBR's query optimizer, and that integrate with Spark's memory manager, monitoring, and I/O systems. By integrating with DBR at the operator level, customers can continue to run their workloads unmodified and obtain the benefits of Photon transparently. Queries can partially run in Photon and fall back to Spark SQL for unsupported operations, while Photon features are being continuously added to reduce these transitions. This ability to partially roll out Photon has given us valuable operational experience in using Photon in the field. Photon also plugs into features like live metrics, so queries that use Photon will show up in the Spark UI just as before. Finally, we rigorously test Photon to ensure that its semantics do not diverge from Spark SQL's, thus preventing unexpected behavior changes in existing workloads.

2 BACKGROUND

To provide context for how Photon fits into a production Lakehouse system, this section describes Databricks' Lakehouse product.

2.1 Databricks' Lakehouse Architecture

Databricks' Lakehouse platform consists of four main components: a raw data lake storage layer, an automatic data management layer to enable warehouse-style transactions and features such as rollback, an elastic execution layer to execute analytics workloads, and a user interface through which customers interact with their data.

Data Lake Storage. Databricks' platform decouples data storage from compute, allowing customers to choose their own low-cost storage provider (e.g., S3, ADLS, GCS). This design prevents customer data lock-in, and also avoids expensive migrations, as customers can connect Databricks to their existing large datasets in cloud storage. Databricks accesses customer data using *connectors* between a compute service and the data lake. The data itself is stored in open file formats such as Apache Parquet.

Automatic Data Management. A majority of Databricks customers have migrated their workloads to use Delta Lake [18], an open source ACID table storage layer over cloud object stores. Delta Lake enables warehouse-style features such as ACID transactions, time travel, audit logging, and fast metadata operations over tabular datasets. Delta Lake stores both its data and metadata as Parquet.

In contrast with Delta Lake, traditional warehouses ingest customer data into a proprietary format in the name of query performance [13, 21]. With Delta Lake, we have found that with the right

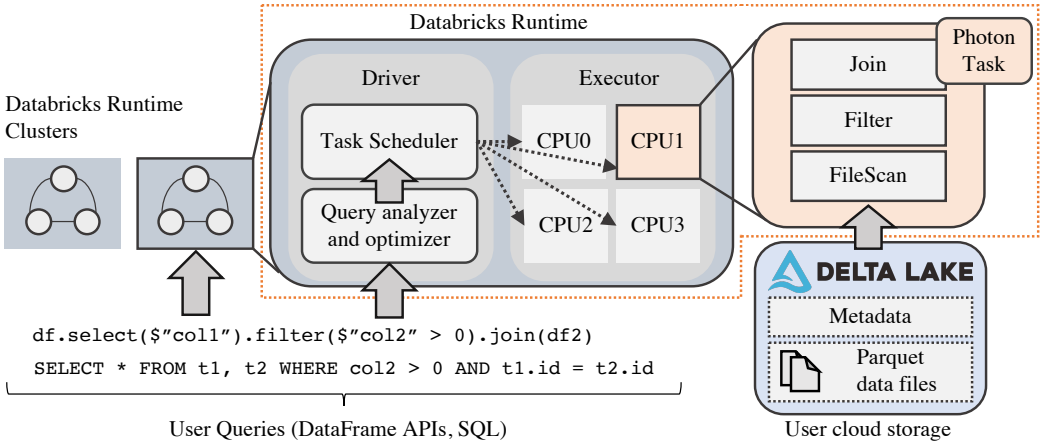

Figure 1: Databricks' execution layer. Photon runs as part of the Databricks Runtime, which executes queries on a distributed cluster of public cloud VMs. Within these clusters, Photon executes tasks on partitions of data on a single thread.

access layer, many of the storage optimizations in proprietary data warehouses are also possible with open file formats. Databricks implements several other optimizations, such as automatic data clustering and caching, on top of the Delta Lake format to further improve performance, e.g., by clustering records based on common query predicates to enable data skipping and reduce I/O [18].

Elastic Execution Layer. Figure 1 illustrates Databricks' execution layer. This layer implements the "data plane" on which all data processing runs. The execution layer is responsible for executing both "internal" queries such as auto data-clustering and metadata access and customer queries such as ETL jobs, machine learning, and SQL. At Databricks, the execution layer reads and processes exabytes of data per day. As a result, this component must be scalable, reliable, and deliver excellent performance in order to reduce costs for our customers and enable interactive data analysis. Photon fits into this execution layer by handling single-threaded query execution on each partition of the data processed.

The execution layer uses cloud computing VMs on AWS, Azure and Google Cloud. Databricks manages these VMs at the granularity of *clusters*, which contain a centralized *driver node* that orchestrates the execution and one or more *executor nodes* that read and process data. These VMs run an execution framework that handles user queries (Databricks Runtime), as well as management software to handle log collection, access control, etc. We discuss the Databricks Runtime in more detail in Section 2.2.

2.2 The Databricks Runtime

The Databricks Runtime (DBR) is the component that handles all query execution (Figure 1). It provides all of Apache Spark's APIs, but contains several performance and robustness improvements on top of the open source codebase. Photon is positioned at the lowest level of DBR, and handles single-threaded task execution within the context of DBR's multi-threaded shared-nothing execution model.

Applications submitted to DBR are called *jobs*, and each job is broken up into stages. A stage represents a part of a job that reads one or more files or data exchanges and ends with either a data exchange or a result. Stages themselves are broken up into individual tasks, which execute the same code on different partitions of data. Stage boundaries in DBR are blocking, i.e., the next stage starts after the previous stage ends. This allows fault tolerance or adaptive execution to occur by replaying stages or re-planning queries at stage boundaries.

DBR uses a single driver node for scheduling, query planning, and other centralized tasks. The driver node manages one or more executor nodes, each of which runs a task execution process to scan data, process it, and produce results. This process is multi-threaded, and contains a task scheduler and a thread pool to execute independent tasks submitted by the driver in parallel.

SQL queries share the same execution framework as all other queries, and can constitute one or more jobs. For example, file metadata queries and SQL sub-queries may all execute as separate jobs within the same overall query. The driver is responsible for converting SQL text or a DataFrame object constructed using the Apache Spark's DataFrame APIs into a *query plan*.

A query plan is a tree of SQL operators (e.g., Filter, Project, Shuffle) that maps to a list of stages. After query planning, the driver launches tasks to execute each stage of the query. Each task runs uses the in-memory execution engine to process data. Photon is an example of such an execution engine; it supersedes the previous engine based on Apache Spark SQL.

2.3 Example: End-to-end SQL Query Execution

Consider the SQL query in Listing 1, which a user submits to Databricks. The query executes against two logical tables `customer` and `orders`, both backed by Delta files in the user's cloud account.

The Databricks service will first route the query to a cluster's driver node. The driver is responsible for creating a query plan, including logical optimizations such as operator re-ordering and physical planning such as choosing join strategies. For the example query, several optimizations apply: if the `orders` table is partitioned by date, we can prune partitions to avoid scanning unnecessary data. If the data is clustered, we may be able to skip additional files if

```
SELECT
  upper(c_name), sum(o_price)
FROM
  customer, orders
WHERE
  o_shipdate > '2021-01-01' AND
  customer.c_age > 25 AND
  customer.c_orderid = orders.o_orderid
GROUP BY
  c_name
```

Listing 1: An example SQL query. This query can benefit from both storage optimizations such as file clustering, as well as runtime engine optimizations that Photon provides, such as SIMD vectorized execution.

the predicates on `age` are known not to match. Databricks supports both these optimizations in its Lakehouse by providing features such as Hilbert clustering [32]. The Delta format additionally makes metadata operations such as listing the files for the latest snapshot of a table fast [18].

Once the driver chooses the files to scan and finalizes a physical query plan, the driver converts the query plan to executable code via the Apache Spark RDD API, and sends the serialized code to each executor node. The executors run this code as tasks on partitions of the input data. These tasks fetch data from the user's cloud storage (or from a local SSD cache, if the user has executed a query over this table before), and then evaluate the remaining query operators.

3 EXECUTION ENGINE DESIGN DECISIONS

In this section, we first provide an overview of Photon's architecture. Then, we dive into the main design decisions in the query engine.

3.1 Overview

Photon is a native (i.e., implemented in C++) execution engine that is compiled into a shared library and invoked from DBR. Photon runs as part of a single-threaded task in DBR, within an executor's JVM process. Like DBR, Photon structures a SQL query as as tree of operators, where each operator uses a HasNext()/GetNext() API to pull batches of data from its child. This API is also used to pull data from operators that are implemented in Java, using the Java Native Interface [8]. Similarly, operators on top of Photon can pull data from it using this same API. Photon also differs from the Java operators because it operates over columnar data, and uses interpreted vectorization instead of code generation to implement its operators: these differences mean that the in-memory data layout that Photon and the Java operators expect may be different. In the remainder of this section, we motivate these differences between the existing and new engines in more detail.

3.2 JVM vs. Native Execution

An early design choice we made was to move away from the JVM and implement the new execution engine in *native code*. This was a significant decision because the existing Databricks Runtime is JVM-based, so shifting to a native engine that integrates with the rest of the runtime was a challenging endeavor.

Our decision to move away from a JVM-based engine was rooted in the observation that our workloads were becoming CPU-bound, and that improving the performance of the existing engine was increasingly difficult. Several factors contributed to this. First, low-level optimizations such as local NVMe SSD caching [38] and auto-optimized shuffle [55] have significantly reduced IO latency. Second, techniques such as data clustering, enabled by Delta Lake, allow queries to more aggressively skip unneeded data via file pruning [32], further reducing IO wait times. Finally, the Lakehouse has introduced new workloads that require heavy data processing over un-normalized data, large strings, and unstructured nested data types: this further stresses in-memory performance.

The consequence of this was that the in-memory execution carried out by JVM-based execution engine was becoming more of a bottleneck, but squeezing more performance out of it requiring heavy knowledge of JVM internals to ensure that the JIT compiler produced optimal code (e.g., loops that used SIMD instructions). Anecdotally, the only engineers that regularly updated the generated Java code were ones that had worked on JVM internals in the past. In addition, we found that the lack of control over lower-level optimizations such as memory pipelining and custom SIMD kernels also contributed to a performance ceiling in the existing engine.

We also found that we were also starting to hit performance cliffs within the JVM on queries in production. For example, we observed that garbage collection performance was seriously impacted on heaps greater than 64GB in size (a relatively small limit given the memory sizes of modern cloud instances). This required us to use manually managed off-heap memory within even the JVM-based execution engine, leading to code that wasn't necessarily easier to write or maintain compared to code written in a native language. Similarly, the existing execution engine, which performs Java code generation [54], was constrained by limits on generated method size or code cache size and would need to fall back to a far slower Volcano-style [31] interpreted code path. For wide tables (e.g., 100s of columns, common in the Lakehouse), we hit this limit regularly in production deployments. In all, after evaluating the engineering effort it would require to sidestep the performance ceilings and scalability limitations of the JVM, we chose to implement a native query execution runtime.

3.3 Interpreted Vectorization vs. Code-Gen

Modern high performance query engines predominantly follow one of two designs: either an *interpreted vectorized* design like in the MonetDB/X100 system [23], or a *code-generated* design like the ones used in Spark SQL, HyPer [41], or Apache Impala [3]. In short, interpreted vectorized engines use a dynamic dispatch mechanism (e.g., virtual function calls) to choose the code to execute for a given input but process data in batches to amortize virtual function call overhead, enable SIMD vectorization, and better utilize the CPU pipeline and memory hierarchy. Code-generated systems do away with virtual function calls by using a compiler at runtime to produce code specialized for the query.

When prototyping our native engine, we tried native implementations of both of the above approaches, using Weld [45] as

a starting point for the code-generating approach. At the end, we chose to proceed with the vectorized approach for a combination of technical and practical reasons, described below.

Easier to develop and scale. One early observation for the code generation approach was that it was harder to build and debug. Since the engine generates the executing code at runtime, we would manually need to inject code that would make finding issues easier. In addition, we found that existing tooling (e.g., debuggers, stack trace tools, etc.) were difficult to use without manually adding instrumentation. In contrast, the interpreted approach was "just C++", for which existing tools are highly tailored. Techniques such as print debugging were also much easier in the interpreted engine.

Interestingly, we found that a majority of the work in using a code generating runtime in the context of a larger system was around adding tooling and observability rather than building the compiler. For example, Weld had a few performance issues that we needed to address before comparing the two approaches, but debugging these issues was difficult without tools such as `perf`. Anecdotally, it took our engineers two months to prototype aggregation with a code-generating engine, and a couple weeks with the vectorized engine.

Observability is easier. Code generation typically eliminates interpretation and function call overheads by collapsing and inlining operators into a small number of pipelined functions. Although this is great for performance, it makes observability difficult. For example, it is challenging to efficiently report metrics on how much time is spent within each query operator given that the operator code may be fused into a row-at-a-time processing loop. The vectorized approach maintains the abstraction boundaries between operators and amortizes overhead by processing batches of data at a time: each operator can thus maintain its own set of metrics. This is especially useful after deploying the engine to customers, since these metrics are the primary interface to debugging performance issues in customer workloads where queries may not be shareable or directly executable by the engine developers.

Easier to adapt to changing data. As we discuss in §4.6, Photon can adapt to batch-level properties by choosing a code-path at runtime. This is particularly important in the Lakehouse context, since traditional constraints and statistics may not be available for all classes of queries. An interpreted-vectorized execution model made adaptivity much easier, since dynamic dispatch is already fundamental to the engine. To achieve the same effect with a code-generating engine, we would have had to either compile a prohibitive number of branches at runtime or re-compile parts of the query dynamically, which would impact query execution time, memory usage, etc. Although this is certainly possible, it incurs additional compilation time and startup overhead. In fact, even HyPer [41], the benchmark for code-generating engines today, includes an interpreter to circumvent these costs under certain scenarios.

Specialization is still possible. Code-generation has clear performance advantages in some scenarios. For example, complex trees of expressions may be simplified using classic compiler optimizations such as common sub-expression elimination, unused column references may automatically be pruned via dead-store elimination, and problems such as sparse batches of data are non-issues since tuples are processed without interpretation overhead.

Despite these advantages, we found that in many cases, we could get similar benefits with our vectorized engine by creating specialized fused operators for the most common cases. For example, we observed that the `between` expression, which evaluates `col >= left and col <= right`, is common in our customers' queries but could lead to high interpretation overhead if expressed as a conjunction. We thus created a special fused operator for it. In general, we felt that the trade-off in complexity allowed us to divert engineering time for these kinds of specializations, which has allowed us to close the performance gap between the two models in many cases.

3.4 Row vs. Column-Oriented Execution

A third design choice we made was to default to a *columnar* in-memory data representation in Photon rather than adopting Spark SQL's row-oriented data representation. In a columnar representation, values of a particular column are stored contiguously in memory, and a row is logically assembled by accessing a specific element within each column.

The advantages of columnar execution have been detailed in prior works [35, 51]. To summarize, columnar layouts are more amenable to SIMD, enable more efficient data pipelining and prefetching by enabling operators to be implemented as tight loops, and can lead to more efficient data serialization for exchanges and spilling. An additional advantage in the Lakehouse context is that our execution engine predominantly interfaces with columnar *file formats* such as Parquet: a columnar representation can thus skip a possibly expensive column-to-row pivoting step when scanning data. Additionally, we can maintain dictionaries to reduce memory usage, which is important especially for string and other variable-length data (again, common in our Lakehouse setting). Finally, writing columnar data is also easier with a columnar engine.

In practice, Photon does pivot columns to rows in certain scenarios. For example, we generally buffer data in data structures such as hash tables as rows, since storing data as columns here requires expensive random accesses when performing operations such as key comparisons during hash table probing.

3.5 Partial Rollout

A final design choice was to enable *partial rollout* of the new execution engine. This meant that the new engine needed to integrate not only with the execution framework (i.e., task scheduling, memory management, etc.), but also the existing SQL engine.

This decision was made for entirely practical reasons. The existing SQL engine, which is based on open-source Apache Spark, is a moving target. The open source community actively contributes improvements, features, and bug fixes regularly. For example, in July 2021, the open source Spark project had 321 commits, of which 47% were to the SQL package. As a result, building a new execution engine that supports *all* the features of the existing one would simply be impossible. The consequence of this is a design that can *partially* execute a query in the new engine, and then gracefully fall back to the old engine for features that are thus far unsupported.

The next two sections describe how the design decisions outlined in this section are realized in our implementation. §4 describes

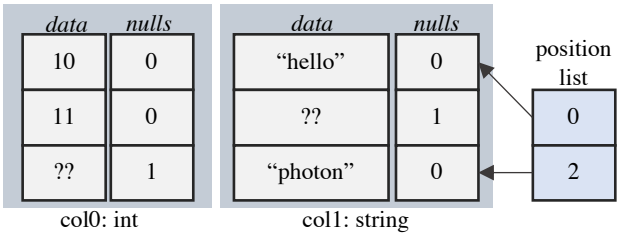

Figure 2: A column batch in Photon. This batch represents the schema *{int, string}* with the tuples *(10, “hello”)* and *(null, “photon”).* Data at inactive row indices may still be valid.

our native, vectorized, column-oriented execution engine, Photon, in more detail. §5 discusses the specific challenges of integrating Photon with our existing JVM-based framework and execution engine, and how we have enabled partial rollout.

4 VECTORIZED EXECUTION IN PHOTON

In this section, we describe the implementation of Photon's vectorized columnar execution engine.

4.1 Batched Columnar Data Layout

Photon represents data in a columnar data format, where each column value is stored contiguously in memory. Groups of columns (which logically form a group of rows) are broken and processed in batches to bound memory usage and exploit cache locality.

The basic unit of data in Photon is thus a single column holding a single batch worth of values, called a *column vector*. In addition to a contiguous buffer of values, the column vector holds a byte vector to indicate the NULL-ness of each value. Column vectors can also hold batch-level metadata such as string encoding.

A *column batch* is a collection of column vectors, and represents rows. Figure 2). In addition to holding a column vector for each column in the row, a column batch contains a *position list* data structure, which stores the indices of the rows in batch that are “active” (i.e., have not been filtered out and should be considered by expressions, operators, etc.). Accessing a row in the column batch requires indirection through the position list, as shown in Listing 2.

Another possible design for designating rows as active vs. inactive is a byte vector. This design is more amenable to SIMD, but requires iterating over all rows even in sparse batches. Our experiments showed that in most cases this led to worse overall performance for all but the simplest queries, since loops must iterate over *O(batch size)* elements instead of *O(active rows)* elements. Recent work confirms our conclusions [42].

Data in Photon flows through operators (e.g., *Project*, *Filter*), at the granularity of column batches. Each operator receives a column batch from its child and produces one or more output batches.

4.2 Vectorized Execution Kernels

Photon's columnar execution is built around the concept of *execution kernels*, which are functions that execute highly optimized loops over one or more vectors of data. This idea was first proposed in the X100 system [23]. Almost all operations on the data plane are implemented as kernels. For example, expressions, probes into a hash table, serialization for data exchange, and runtime statistics calculation are all implemented as kernels at the lowest level. These kernels can sometimes use hand-coded SIMD intrinsics, but often we rely on the compiler to auto-vectorize the kernel (and provide hints such as `RESTRICT` annotations [1] on the inputs to aid in auto-vectorization). Kernels can be specialized for different input types using C++ templates.

Photon invokes operators and expressions at the granularity of vectors. Each kernel takes vectors and the column batch position list as input and produces a vector as output. Operators pass vectors among to other operators until eventually being passed out of Photon for external use (e.g., by Spark). Listing 2 shows an example kernel, for an expression that computes the square root of a value.

4.3 Filters and Conditionals

Filters in Photon are implemented by modifying the position list of a column batch. A filtering expression takes column vectors as input and returns a position list (which is a subset of the input position list) as output. To implement conditional expressions such as `CASE WHEN`, we modify the position list so only some rows are “turned on” within the kernel invocation for each branch, while writing to the same output vector. Note that this means that modifying inactive rows (e.g., to enable SIMD) is disallowed, since inactive row positions may still contain valid data.

4.4 Vectorized Hash Table

Unlike a standard scalar-access hash table, Photon's hash table is optimized for vectorized access. Lookups to the hash table occur in three steps. First, a hash function is evaluated on a batch of keys using a hashing kernel. Next, a probe kernel uses the hash values to load pointers to hash table entries (entries in the hash table are stored as rows, so a single pointer can represent composite keys). Finally, the entries in the hash table are compared against the lookup keys column-by-column, and a position list is produced for non-matching rows. Non-matching rows continue probing the hash table by advancing their bucket index for filled buckets according to the probing strategy (we use quadratic probing).

Each step benefits from vectorized execution. The hashing and key comparison benefit from SIMD. Since the probe step issues random memory accesses, it also benefits from vectorization. The independent loads are close to each other in the kernel code, and can therefore be parallelized by the hardware. Note that other designs might be able to achieve similar effects via software prefetching, but in our experience, manual prefetching is difficult to tune and tends to have different optimal configurations for different hardware.

4.5 Vector Memory Management

Memory management is an important consideration in any execution engine. To prevent expensive OS-level allocations, Photon allocates memory for transient column batches using an internal buffer pool, which caches allocations and allocates memory using a most-recently-used mechanism. This keeps hot memory in use for repeated allocations for each input batch. Since the query operators are fixed during execution, the number of vector allocations required to process a single input batch end-to-end is fixed.

```
template <bool kHasNulls, bool kAllRowsActive>
void SquareRootKernel(const int16_t* RESTRICT pos_list,
  int num_rows, const double* RESTRICT input,
  const int8_t* RESTRICT nulls, double* RESTRICT result) {
  for (int i = 0; i < num_rows; i++) {
    // branch compiles away since condition is
    // compile-time constant.
    int row_idx = kAllRowsActive ? i : pos_list[i];
    if (!kHasNulls || !nulls[row_idx]) {
      result[row_idx] = sqrt(input[row_idx]);
    }
  }
}
```

Listing 2: A Photon kernel that is specialized on the presence of NULLs and inactive rows. Branches over compile-time constant template parameters are optimized away.

Variable length data (e.g., buffers for strings) is managed separately, using an append-only pool that is freed before processing each new batch. Memory used by this pool is tracked by a global memory tracker, so the engine could in theory adjust the batch size if it encounters large strings that it cannot accommodate.

Large persistent allocations that outlive any single batch (e.g., for aggregations or joins) are tracked separately using an external memory manager. We discuss these allocations in more detail in §5. We have found fine-grained memory allocation to be valuable because, unlike the Spark SQL engine, we can more robustly handle large input records that are frequent in our Lakehouse setting.

4.6 Adaptive Execution

A major challenge in the Lakehouse context is the lack of statistics, metadata, or normalization of the query input. While some tables will have statistics available for planner decisions, others will have no information beyond the schema. In addition, NULL data is common, and expressions that manipulate and normalize strings also appear frequently. The execution engine thus bears the burden of efficient execution on queries over such data. To solve this, our engine supports *batch-level adaptivity*. In short, Photon can at runtime build metadata about a batch of data and use it to optimize its choice of execution kernel.

Every execution kernel in Photon can adapt to at least two variables: whether there are any NULLs in the batch, and whether there are any inactive rows in the batch. Lack of NULLs allows Photon to remove branching, which improves performance. Similarly, if there are no inactive rows in a batch, Photon can avoid the indirect lookup via the position list, which again improves performance and enables SIMD. Listing 2 shows an example of these specializations.

Photon specializes several other kernels on a case-by-case basis. For example, many string expressions can be executed with an optimized code path if the strings are all ASCII encoded (as opposed to general UTF-8). Photon can thus compute and store metadata about ASCII-ness within a column vector and use this information to choose the correct kernel at runtime. As another example, Photon

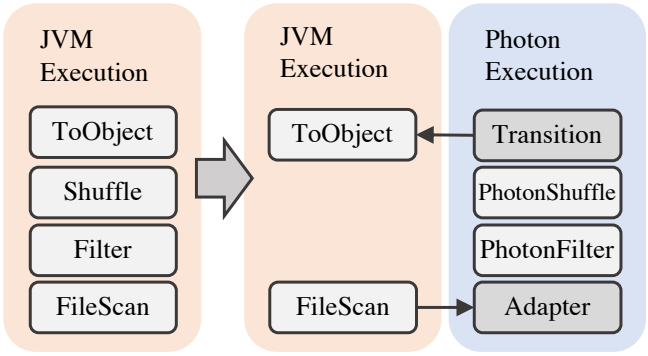

Figure 3: A Spark SQL plan converted to a Photon plan.

can selectively choose to *compact* batches that are sparse at runtime in order to improve performance. This is especially impactful when probing large hash tables using a vectorized API, since dense batches can better exploit memory parallelism by issuing loads from the hash table in parallel; on the other hand, sparse batches incur high memory latency without saturating memory bandwidth.

We have also explored adaptivity in other scenarios, e.g., adaptive shuffle encoding by finding patterns in user data at runtime. For example, we found that many users encode unique identifiers as 36-character strings instead of (an equivalent) 128-bit integer. We have also observed numeric data (e.g., integers) encoded as strings, which can be serialized more efficiently using a binary format.

5 INTEGRATION WITH DATABRICKS RUNTIME

Photon integrates with the Databricks Runtime (DBR). Unlike a traditional data warehouse, Photon shares resources with all other workloads that execute over DBR and the Lakehouse storage architecture, ranging from data cleaning for machine learning models to ETL. In addition, Photon co-exists with the old Spark SQL-based execution engine for queries with operators that do not yet support the new execution engine. For these reasons, Photon must integrate tightly with DBR's query planner and memory manager.

5.1 Converting Spark Plans to Photon Plans

To integrate with the legacy Spark SQL-based engine, we convert a physical plan that represents execution using the legacy engine into one that represents execution with Photon. This transformation is done via new rule in Catalyst [20], Spark SQL's extensible optimizer. A Catalyst rule is a list of pattern matching statements and corresponding substitutions that are applied to a query plan. If a pattern matches on any query plan node, that node is replaced with the corresponding substitution. Figure 3 shows an example.

The rule proceeds as follows. First, we walk the input plan bottom up, starting at file scan nodes, and map each supported legacy engine node to a Photon node. When we see a node that Photon does not support, we insert a *transition node* that converts the columnar Photon format to the row-wise format that the legacy engine uses. We do not transform nodes starting in the middle of the plan to avoid regressions from too many column-to-row pivots. We also add an *adapter* node between the file scan and the first

Photon node: this maps the legacy scan input to Photon columnar batches, but is zero-copy since the scan produces columnar data.

5.2 Executing Photon Plans

After query planning, DBR launches tasks to execute the stages of the plan. In a task with Photon, the Photon execution node first serializes the Photon part of the plan into a Protobuf [6] message. This message is passed via the Java Native Interface (JNI) [8] to the Photon C++ library, which deserializes the Protobuf and converts it into a Photon-internal plan. Internally, the execution plan in Photon looks similar to its DBR counterpart: each operator is a node with a `HasNext()/GetNext()` interface, and data is pulled (at the granularity of column batches) by a parent node from the child node. Note that Photon runs in the JVM process and communicates with the Java runtime using JNI.

For plans that end with a data exchange, Photon writes a shuffle file that conforms to Spark's shuffle protocol, and passes metadata about the shuffle file to Spark. Spark then performs the shuffle using this metadata, and a new Photon task (in a new stage) reads the relevant partitions from the shuffle. Since we use a custom data serialization format that is not compatible with Spark's format, a Photon shuffle write must be accompanied by a Photon shuffle read.

Adapter node to read Scan data. The leaf node in a Photon plan is always an "adapter" node. The Photon adapter node works by taking columnar data produced by Spark's scan node and passing pointers to this data to Photon. Within Photon, the adapter node's `GetNext()` method makes a C++ to Java JNI call that passes a list of native pointers to the JVM. We pass two pointers per column: one to represent the vector of column values, and one to represent the NULL values for each column value. On the Java side, the scan node directly produces columnar data that is stored in off-heap memory via the open-source `OffHeapColumnVector` [9] class in Spark. Like Photon, this class stores values as primitives back-to-back in off-heap memory, and stores NULLs as an off-heap byte array (one byte per value). Thus, the adapter node just needs to take the pointers provided by Photon and point them to the off-heap column vector memory without copying. We make one JNI call per batch to consume the scan data. We note that, in our measurements, the overhead of making a JNI call is comparable to a C++ virtual function call (roughly 23ns per call).

Transition node to pass Photon data to Spark. The last node in a Photon plan is a "transition"node. Unlike the adapter node, the transition node must pivot columnar data to row data so the row-wise legacy Spark SQL engine can operate over it. Since Apache Spark's scan always produces columnar data when reading columnar formats, we note that one such pivot is required *even without Photon*. Since we only convert plans to Photon starting at the scan node, adding a single pivot on top of a Photon plan does not cause regressions vs. Spark (both the Spark plan and Photon plan each have a single pivot). However, if we were to eagerly convert arbitrary parts of the plan to use Photon, we could have an arbitrary number of pivots, which could lead to regressions. Today, we elect to be conservative and choose not to do this. In the future, we may investigate weighing the tradeoff of the speedup Photon would provide vs. the slowdown caused by adding an additional column-to-row pivot.

5.3 Unified Memory Management

Photon and Apache Spark share the same cluster and thus must have a consistent view of memory and disk usage to avoid being OOM-killed by the OS or the JVM. As a result, Photon hooks into Apache Spark's memory manager.

To handle this, we separate the concept of memory *reservations* from *allocations* in Photon. A memory reservation asks for memory from Spark's unified memory manager. Like all requests to the memory manager, this can cause a *spill*, where Spark asks some memory consumer to release memory to satisfy a new request. Photon hooks into this memory consumer API, so Spark can ask Photon to spill data on behalf of other memory-consuming operators that are executing (e.g., a sort task in Spark may ask a Photon operator to spill). Similarly, Photon can make reservations that cause other Spark operators to spill, or cause Photon itself to spill (leading to a "recursive spill" where one Photon operator spills memory on behalf of another one). This differs slightly from many other database engines, where operators are given a fixed memory budget and can only "self-spill." Spilling is dynamic because we often do not have information on how much data an operator will consume, especially if SQL operators are co-existing with user-defined code that also reserves memory.

We use the same policy as open source Apache Spark to determine which operator to spill. To summarize, if we need to spill N bytes to satisfy a memory reservation request, we sort the memory consumers from least to most allocated memory, and spill the first consumer that holds at least N bytes. The rationale behind this is that we minimize the number of spills and avoid spilling more data than necessary.

After reserving memory, Photon can allocate memory safely without spilling. Allocation is purely local to Photon: Spark only accounts for the memory that Photon asks for. In spilling operators such as hash join and grouping aggregation the processing of an input batch is thus broken up into two phases: a reservation phase where memory is acquired for the new input batch and spilling is handled, and an allocation phase where transient data can be produced since no spilling can occur. This flexible spilling mechanism has been critical in the Lakehouse context, where queries often exceed the available memory.

5.4 Managing On-heap vs. Off-heap memory

Both DBR and Apache Spark support requesting off-heap and on-heap memory from the memory manager. To manage off-heap memory, the Spark cluster is configured with a static "off-heap size" per node, and the memory manager is responsible for handing out memory from this allocation. It is the responsibility of each memory consumer to only use the memory allocated; overuse can lead to OOM-kills by the operating system.

Unfortunately, just provisioning memory is not sufficient for stable operation. The JVM usually performs garbage collection when it detects high on-heap memory usage. However, with Photon, most of the memory usage is off-heap, so garbage collection seldom occurs. This is problematic if Photon relies on on-heap memory for parts of its query. One example of this is with *broadcasts*. Photon uses Spark's builtin broadcasting mechanism to share data with each node in a cluster (e.g., for broadcast hash join). The broadcast

mechanism is implemented in Java, so it requires a copy from off-heap Photon memory to on-heap Spark memory. However, this transient memory is not garbage collected frequently, and can lead to Java OutOfMemory errors if some other Spark code tries to make a large allocation. We solved this by adding a listener that cleans up Photon-specific state after the query terminates: this ties Photon state to the lifetime of a query instead of a GC generation.

5.5 Interaction with Other SQL Features

Photon represents a plan change in a DBR query. Despite this, many of the non-trivial performance features in both DBR and Apache Spark are compatible with Photon. For example, Photon can participate in adaptive query execution [29], in which runtime statistics are used to re-partition and re-plan a query at runtime at the stage boundaries. Photon's operators implement the interfaces required to export statistics for such decisions (e.g., the size of shuffle files and the number of output rows produced at the end of a stage). Similarly, Photon can take advantage of optimizations such as shuffle/exchange/subquery reuse and dynamic file pruning, all features that are particularly critical in the Lakehouse context to enable efficient data skipping. Finally, Photon also supports integration with Spark's metrics system, and can provide live metrics that appear in the Spark UI during query execution.

5.6 Ensuring Semantics Consistency

Another interesting challenge we faced was ensuring that Photon's *behavior* was identical to Apache Spark's. This is because the same query expression can run in either Photon or Spark depending on whether some other part of the query was able to run in Photon, and the results must be consistent. For example, Java and C++ implement integer-to-floating point casts differently, which can lead to different results in some scenarios. As another example, many time-based expressions rely on the IANA timezone database [14]. The JVM ships with a particular version of this database, and if Photon were to use a different version, many time-based expressions would return different results. To check results against Spark, we use three kinds of testing: *(1)* unit tests that explicitly enumerate test cases for, e.g., SQL expressions, *(2)* end-to-end tests that explicitly compare results vs. Spark on SQL queries, and *(3) fuzz tests* that randomly generate input data and compare the results with Spark. We discuss each below.

Unit tests. We use two kinds of unit tests. In native code, we have built a unit testing framework for SQL expressions (e.g., upper(), sqrt(), etc.) that allows specifying input and output values for any expression in a table: the framework then loads the table into column vectors, evaluates the expression on all the available specializations (e.g., no NULLs, with NULLs, no inactive rows, etc.) and compares the result to the expected output. This also ensures that inactive rows are not incorrectly overwritten.

We also integrate with Spark's existing open source expression unit tests. These tests hook in with our function registry, which determine whether the test case is supported in Photon. If it is, we compile a query for the unit test (e.g., `SELECT expression(inputs) FROM in_memory_table`) and execute it against both Spark and Photon, and then compare the results. This gives us the full unit test coverage contributed by the open source community and Databricks.

End-to-end tests. End-to-end tests test query operators by submitting a query against Spark and Photon and comparing the results. We have a dedicated set of tests that are executed only when Photon is enabled (e.g., to test out-of-memory behavior or certain Photon-specific plan transformations), but we also enable Photon on the full suite of Spark SQL tests for additional coverage. We have found several "unexpected" bugs this way, e.g., memory corruption caused by enabling off-heap memory in Spark and producing incompatible/corrupt data from the file scan.

Fuzz tests. A final source of compatibility testing comes from fuzz testing, in which we test random queries against Spark and Photon. Our fuzz testing consists of a fully general random data and query generator as well as surgical fuzzers for specific features.

One specialized fuzz tester we use is for Photon's decimal implementation. Photon's decimal differs in behavior from Spark, in that it can operate over inputs of different types for performance (in contrast, Spark always casts its decimal inputs to the output type first, which can leads to worse performance). This difference leads to some unavoidable differences in behavior, which our fuzz tester checks for by using a behavior whitelist.

6 EXPERIMENTAL EVALUATION

In this section, we experimentally demonstrate Photon's speedups over DBR. In particular, we seek to answer the following questions:

(1) What query shapes benefit the most from Photon?

(2) How does Photon perform end-to-end vs. our existing engine?

(3) What is the impact of tactical optimizations like adaptivity?

6.1 Which Queries will Photon Benefit?

Photon primarily improves the performance of queries that spend a bulk of their time on CPU-heavy operations such as joins, aggregations, and SQL expression evaluation. Queries with these operations are most impacted by Photon's differences over DBR: native code, columnar, vectorized execution, and runtime adaptivity. Photon can also provide speedups on other operators such as data exchanges and writes by either speeding up the in-memory execution of these operators or by using a better encoding format when transferring data over the network. We do not expect Photon to significantly improve the performance of queries that are IO or network bound.

We first compare Photon vs. DBR on micro-benchmarks for queries we expect Photon to benefit: joins, aggregations, expression evaluation, and Parquet writes. we then show how these improvements translate to end-to-end query performance on TPC-H.

Micro-benchmark Setup. We run our micro-benchmark experiments on a single `i3.2xlarge` Amazon EC2 instance with 11GB of memory for the JVM and 35GB of off-heap memory for Photon[1]. The benchmarks run on a single thread. We read from an in-memory table to isolate the effects of Photon's execution improvements.

Hash Joins. To benchmark join performance, we compare Spark's sort merge join and hash join vs. Photon's hash join[2]. We create two artificial tables with 1GB of integer data each and perform an inner equi-join over the integer columns.

[1] This is the default setting on this instance type on Photon-enabled clusters.
[2] Photon does not support sort-merge join, but Apache Spark uses it by default because its shuffled hash join implementation does not support spilling.

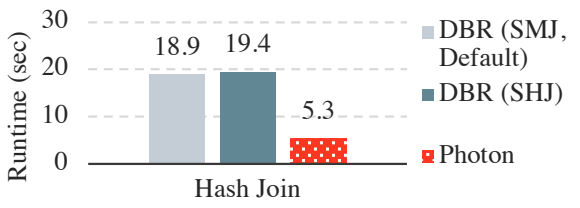

Figure 4: `SELECT count(*)from t1, t2 WHERE t1.id = t2.id`. The join runs 3× faster in Photon with the vectorized hash table. We compare against DBR's sort-merge join (SMJ) and shuffled-hash join (SMJ).

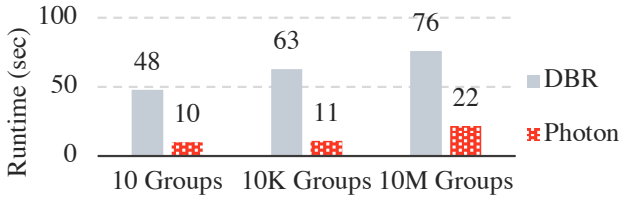

Figure 5: `SELECT collect_list(strcol)GROUP BY intcol`. Native memory management and a fast vectorized hash table produce an up to 5.7× speedup.

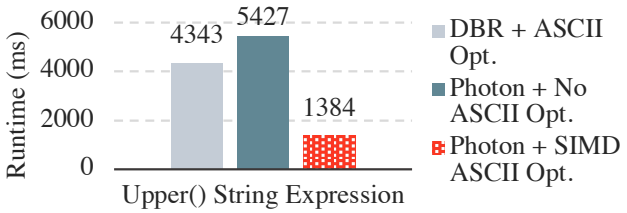

Figure 6: Performance benefits of vectorization and native code, exemplified by a custom SIMD ASCII check kernel. The custom kernel provides a 3× speedup over DBR.

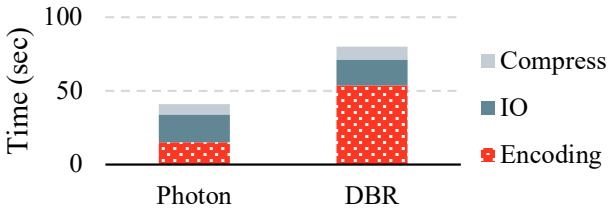

Figure 7: Benefits of using Photon to write Parquet. Optimized column encoders provide a 2× speedup.

Figure 4 shows the results. Photon's vectorized hash table outperforms DBR by 3.5×, primarily due to better memory hierarchy utilization by parallelizing loads: an optimization facilitated by interpreted-vectorized execution. Photon also reduces memory allocation churn by leveraging the vector buffer pool, leading to further speedups. Although Photon also uses SIMD for hashing and key comparison, most joins are memory-intensive, so the improvements to load parallelism during probing have the most impact on performance.

Aggregations. To demonstrate the benefits of Photon on aggregation queries, in this benchmark we run a grouping aggregation on a string column on a varying number of integer groups with the CollectList aggregation function. This function collects input rows into an array data type. The DBR implementation of this function uses Scala collections to perform the aggregation and does not support code generation, in large part because the Spark SQL code generation framework is incompatible with aggregation functions that require variable size aggregation states. Photon uses a custom vectorized implementation of the expression.

Figure 5 shows the results. Photon outperforms DBR on this microbenchmark by up to 5.7×. Like the join, Photon benefits from a vectorized hash table while resolving the groups to aggregate into. Photon also employs techniques such as memory pooling to improve performance while evaluating the aggregation expression itself, by coalescing allocations for lists across groups instead of managing the state for each group independently.

Expression Evaluation. To show the impact of evaluating expressions in Photon, we ran a benchmark that runs the `upper` expression to uppercase a string. Both DBR and Photon special-case the upper-casing expression for ASCII strings: ASCII strings can be uppercased using a simple byte-wise subtraction operation, but general UTF-8 strings require use of a Unicode library that maps each UTF-8 codepoint to its uppercased equivalent. In Photon the string ASCII check and the upper-casing kernel both use SIMD.

Figure 6 shows the results. As a baseline, we also show Photon without specializing for ASCII at all: each string is uppercased using the ICU [7] library (the same one DBR would use for its UTF-8 codepath). Photon's custom SIMD ASCII check and the upper-casing kernel together contribute to a 3× speedup over DBR, and a 4× speedup over the ICU-based implementation.

Overall, Photon supports over 100 expressions from Apache Spark. Expression evaluation in Photon benefits heavily from native code and buffer reuse via the buffer pool, and is especially impactful in cases where DBR relies on generic Java libraries for computation.

Parquet Writes. Photon supports writing Parquet/Delta files. This operation is used for creating new tables or appending to existing ones. DBR performs this operation using the open source Java-based Parquet-MR library [11]. To compare Photon's performance on Parquet writes vs. DBR, We ran a benchmark that writes a 200M row Parquet table with six columns (integers, longs, dates, timestamps, strings, and booleans). These types cover the most popular Parquet column encodings [10]. Unlike the other benchmarks, to show the impact of writing to cloud storage we run this benchmark on a single-node `i3.2xlarge` cluster with eight threads and 16 partitions, with the table stored in S3. Figure 7 shows the result with a breakdown in runtime. Photon outperforms DBR by 2× end-to-end, with the main speedup coming from the column encoding. This is from a faster hash table to dictionary encode [5] the string columns, and also by using optimized bit-packing [12] and statistics computation kernels. Both DBR and Photon spend roughly the same amount of time compressing pages and writing to storage.

6.2 Comparison vs. DBR on TPC-H

We now evaluate end-to-end query performance on a cluster using the queries from the TPC-H benchmark. We ran the 22 TPC-H queries on an 8-node AWS cluster and one driver node. Each node is an `i3.2xlarge` instance with 64GB of memory and 8 vCPUs (Intel Xeon E5 2686 v4). We ran the benchmark with SF=3000 with the Delta format, stored in Amazon S3. Figure 8 shows the minimum time across three runs for all queries, after a warm-up run.

Overall, Photon achieves a maximum speedup of 23×, and an average speedup of 4× across all queries. Since Photon and DBR use identical logical plans during execution, to better understand the source of speedups we separately profiled one run of each query using a Databricks-internal cluster profiling tool. The profiles show that speedups predominantly come from the operations discussed in §6.1: faster joins and aggregations due to the vectorized hash table, faster expression evaluation, and in some cases, more efficient serialization during shuffles. Most queries are bottlenecked on either a large join or aggregation.

Q1 shows the largest speedup of all queries—23× faster than DBR—because it is heavily bottlenecked on Decimal arithmetic. Photon vectorizes Decimal arithmetic with native integer types. DBR only uses native integers for decimals with low precision, but uses infinite-precision Java Decimal for all other operations, leading to an over 80× speedup on some expressions. Simple scan-and-aggregate queries such as Q6 exhibit speedups for the same reason; aggregations in Photon are vectorized by using native kernels.

Q9 is a representative query bottlenecked on a large join (executed as a shuffled hash join in Photon). We observe that the join operator consumes fewer cycles in Photon than in DBR (compared to both DBR's shuffled hash join and sort-merge join). Most of this speedup comes from masking cache miss latency by parallelizing loads to the hash table, though smaller optimizations like avoiding copies during hash table resizing also contribute to the speedup. With these optimizations, Q9 in Photon is dominated by writing files for data shuffles (50% of the total time).

TPC-DS Performance. As further validation of Photon's performance, the TPC council independently ran the full TPC-DS benchmark on DBR with Photon enabled, on the 100TB scale factor. The data was stored in Amazon S3, and the queries ran on a 256-node cluster of `i3.2xlarge` instances on AWS. The full benchmark includes a power run with all queries run back-to-back, a concurrency run, and a database update run. As of February 2022, DBR with Photon holds the TPC world record on the overall benchmark [15, 53]. Photon speeds up TPC-DS for many of the same reasons as TPC-H: faster expression evaluation, joins, and aggregations.

6.3 Overhead of JVM Transitions

Photon uses the JNI to interface with Spark and also transition operators to pass data between Spark and Photon. To measure the overhead of these operators, we ran a simple query that reads a single integer column from an in-memory table. This shows the worst case overhead of the transition nodes since the query does no other work, and Photon must convert all rows from the Photon representation to the Spark representation. We observed that 0.06% of the execution time was spent in JNI-internal methods, and 0.2% of the time was spent in the adapter node feeding into Photon. The rest of the profile was identical for Photon and DBR: roughly 95% of the time was spent serializing rows into Scala objects so they could be processed with a no-op UDF, and the remaining time went into various Spark iterators. We found no additional overhead from the column-to-row operation, since Spark must apply this operation as well. In all, we have not found the JNI or the transition nodes to be a significant source of overhead, especially when these calls are amortized via batching.

Configuration	Runtime (ms)	Data Size (MB)
DBR	31501	1759.6
Photon + No Adaptivity	17324	1715.1
Photon + Adaptivity	15069	763.2

Table 1: Impact of adaptive UUID encoding in Photon on artificial data. Adaptivity reduces the amount of data LZ4 needs to compress and can reduce spilling.

6.4 Benefits of Runtime Adaptivity

In this section, we run microbenchmarks to show the impact of adaptivity in Photon. We demonstrate two kinds of adaptivity.

Adapting to Sparse Batches. Photon compacts column batches before probing hash tables in order to increase memory parallelism. This feature is adaptive because Photon tracks the sparsity of column batches at runtime. To illustrate its benefits, we ran TPC-DS query 24 on a 16-node cluster with and without adaptive join compaction. Figure 9 shows the results. Overall, compaction produces a 1.5× improvement compared to probing the sparse batches directly. In addition, we note that Photon without compaction regresses vs. DBR: this is a consequence of the vectorized execution model, since sparse column batches additionally cause high interpretation overhead in downstream operators after the join. DBR's code-generation model does not incur this interpretation overhead by nature, since the engine processes value tuple at a time, but "inlines" operators to eliminate overheads. This optimization shows that carefully engineering a vectorized execution engine can also eliminate such overheads, but requires additional specialization.

Adaptive String Encoding Microbenchmark. Data shuffles are still a dominant bottleneck in many queries. In addition to consuming network bandwidth, large shuffles also cause memory pressure, which can lead to spilling and thus reduced performance.

One adaptive mechanism we prototyped in Photon encodes UUID strings as 128-bit integers. UUIDs are a standardized concept [37] and have a canonical 36-byte string format. They appear frequently in the data lake, and are commonly generated directly from Apache Spark as well. Because of their canonical format, UUIDs can be represented using a compact 128-bit representation. Photon can detect string columns with UUIDs before writing a shuffle file and switch to this optimized encoding on demand.

To evaluate this feature, we ran a single-machine micro-benchmark that repartitions a dataset with a UUID string column with 50M rows. We measured the end-to-end execution time and the reduction in shuffle data volume. All schemes compress the data using LZ4 compression before writing the shuffle file. Table 1 shows the

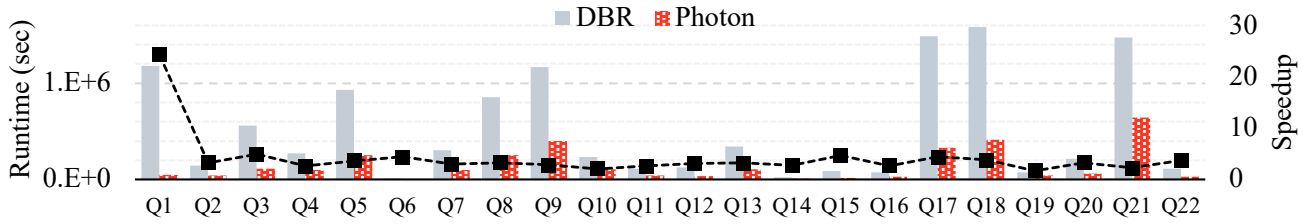

Figure 8: TPC-H performance of Photon vs. DBR (SF=3000). Photon speeds up DBR by an average of 4× per query.

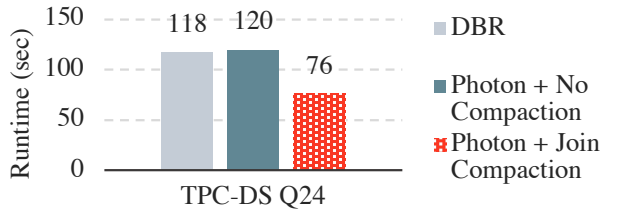

Figure 9: Impact of adaptive join compaction in Photon on TPC-DS Q24. Compaction allows vectorization to outperform code-gen and produces a 1.55× speedup over DBR.

results. Overall, we see a modest 15% reduction in end-to-end execution time on this microbenchmark, primarily from reducing the amount of data that LZ4 needs to compress. However, the new compression scheme leads to an over 2× reduction in data volume. In memory-intensive queries, this reduction can avoid spilling data to disk or causing memory pressure on downstream operators, which can have an outsized impact on end-to-end performance.

7 RELATED WORK

Photon builds on a rich lineage of academic and industrial work on database engine design. Photon's vectorized execution model was introduced by MonetDB/X100 [23], and industrial OLAP DBMSes such as C-Store [51], Vertica [35] and Snowflake [26] have helped prove its practicality.

The batch-level adaptivity scheme described in this paper is similar to micro-adaptivity in the Vectorwise system [47], where queries adapt at a fine-grained scale to changing data. The position list-based representation of row filters was concurrently explored by Ngom et al. [42]: this work reached similar conclusions to our experiments and showed that positions lists outperform byte vectors for all but the simplest queries. The reliance on SIMD for better performance in database operators has also been discussed before [46, 59]. For example, Photon's hash table contains similar access to patterns to Polychroniou et al. [46], though our table contains additional optimizations to fully utilize memory bandwidth.

We discussed our reasons for choosing a vectorized execution model as opposed to a code generating execution model. Several other systems have implemented and evaluated the code generation model, the most notable being HyPer [41]. Prior work [34, 50] has also explored this topic, and some systems have experimented with a hybrid approach [36, 40]. Our engine specializes execution kernels to common patterns as a form of "offline" code-gen, but we have observed that code-gen is likely to help for complex expressions. We consider hybrid approaches future work.

Photon is embedded in the Databricks Runtime, which is based on Apache Spark [57] and Spark SQL [20]. Several other systems have explored optimizing SQL query execution within a more general MapReduce-style framework [17, 24, 27, 28, 30, 44, 48, 56]. Flare [28] in particular embeds a native execution engine within Spark, but does not discuss issues such as memory management and spilling. The Apache Arrow project [2] provides an in-memory format similar to Photon's column vectors, and contains a similar concept of kernels to execute expressions over Arrow buffers.

Photon addresses some of the query processing challenges specific to the Lakehouse storage architecture [19]. End-to-end efficiency is made possible by also addressing challenges related to data storage, data management, and efficient file retrieval, which the Delta Lake [18] storage layer and columnar Parquet file format [4] enable. Many of the optimizations specific to Parquet and Delta have been discussed by Abadi et al. [16], e.g., maintaining column compression from the file format throughout the execution engine.

Finally, Photon was motivated by several long-standing observations: that hardware is becoming increasingly parallel, I/O speeds have outpaced the performance of a single core, and optimizing for the memory hierarchy is critical for performance [22, 25, 33, 39, 43, 52]. These trends will only become important in the future.

8 CONCLUSION

We have presented Photon, a new vectorized query engine for Lakehouse environments that underlies the Databricks Runtime. Photon's native design has solved many of the scalability and performance issues that we had faced with our previous JVM-based execution engine. Its vectorized processing model enables rapid development, rich metrics reporting, and micro-adaptive execution for handling the unstructured "raw" data that is ubiquitous in data lakes. Our incremental rollout has allowed us to enable Photon for hundreds of customers and millions of queries, even in scenarios where the full query is not yet supported. Through standardized benchmarks and micro-benchmarks, we have shown that Photon achieves state-of-the-art performance on SQL workloads.

REFERENCES

[1] 2017. Restrict-qualified pointers in LLVM. https://llvm.org/devmtg/2017-02-04/Restrict-Qualified-Pointers-in-LLVM.pdf.
[2] 2018. Apache Arrow. https://arrow.apache.org/.
[3] 2021. Apache Impala. https://impala.apache.org/.
[4] 2021. Apache Parquet. https://parquet.apache.org.

[5] 2021. Dictionary encoding. https://github.com/apache/parquet-format/blob/master/Encodings.md#dictionary-encoding-plain_dictionary--2-and-rle_dictionary--8.

[6] 2021. Google Protocol Buffers. https://developers.google.com/protocol-buffers/.

[7] 2021. ICU - International Components for Unicode. https://icu.unicode.org/.

[8] 2021. JNI APIs and Developer Guides. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[9] 2021. OffHeapColumnVector. https://github.com/apache/spark/blob/master/sql/core/src/main/java/org/apache/spark/sql/execution/vectorized/OffHeapColumnVector.java.

[10] 2021. Parquet Encodings. https://github.com/apache/parquet-format/blob/master/Encodings.md.

[11] 2021. Parquet-MR. https://github.com/apache/parquet-mr.

[12] 2021. RLE/Bit-packing encoding. https://github.com/apache/parquet-format/blob/master/Encodings.md#run-length-encoding--bit-packing-hybrid-rle--3.

[13] 2021. Snowflake Database Storage. https://docs.snowflake.com/en/user-guide/intro-key-concepts.html#database-storage.

[14] 2021. Time Zone Database. https://www.iana.org/time-zones.

[15] 2021. TPC-DS Result Details. http://tpc.org/tpcds/results/tpcds_result_detail5.asp?id=121103001.

[16] Daniel Abadi, Peter Boncz, Stavros Harizopoulos Amiato, Stratos Idreos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-oriented Database Systems.* Now Hanover, Mass.

[17] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop. https://databricks.com/blog/2016/05/23/

[18] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja undefineduszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (August 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[19] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. CIDR.

[20] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proc. ACM SIGMOD* (Melbourne, Victoria, Australia). 1383–1394. https://doi.org/10.1145/2723372.2742797

[21] Benoit Dageville. 2021. Striking a balance with 'open' at Snowflake. https://www.infoworld.com/article/3617938/striking-a-balance-with-open-at-snowflake.html.

[22] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, Vol. 99. 54–65.

[23] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[24] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2010. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endow.* 3, 1–2 (September 2010), 285–296. https://doi.org/10.14778/1920841.1920881

[25] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 339–351.

[26] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[27] J. Dees and P. Sanders. 2013. Efficient many-core query execution in main memory column-stores. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. 350–361. https://doi.org/10.1109/ICDE.2013.6544838

[28] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-up Architectures and Medium-Size Data. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 799–815.

[29] Wenchen Fan, Herman van Hövell, and MaryAnn Xue. 2020. Adaptive Query Execution: Speeding Up Spark SQL at Runtime. https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[30] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: All for One, One for All in Data Processing Systems. In *Proc. ACM EuroSys* (Bordeaux, France). Article 2, 16 pages. https://doi.org/10.1145/2741948.2741968

[31] Goetz Graefe. 1990. *Encapsulation of Parallelism in the Volcano Query Processing System.* Vol. 19. ACM.

[32] Adrian Ionescu. 2018. Processing Petabytes of Data in Seconds with Databricks Delta. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html.

[33] A Kagi, James R Goodman, and Doug Burger. 1996. Memory Bandwidth Limitations of Future Microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*. IEEE, 78–78.

[34] 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. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

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

[36] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[37] P. Leach, M. Mealling, and R. Salz. 2005. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. https://datatracker.ietf.org/doc/html/rfc4122.

[38] Alicja Luszczak, Michał Szafrański, Michał Switakowski, and Reynold Xin. 2018. Databricks Cache Boosts Apache Spark Performance. https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html.

[39] John D McCalpin et al. 1995. Memory bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)* 1995 (1995), 19–25.

[40] 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.

[41] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[42] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, and Andrew Pavlo. 2021. Filter Representation in Vectorized Query Execution. 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 6, 7 pages. https://doi.org/10.1145/3465998.3466009

[43] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*, Vol. 15. 293–307.

[44] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. 2018. Evaluating End-to-end Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1002–1015.

[45] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[46] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[47] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1231–1242.

[48] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proc. ACM SOSP* (Farmington, Pennsylvania, USA). ACM, 49–68. https://doi.org/10.1145/2517349.2522715

[49] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[50] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[51] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[52] Wm. A. Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.

[53] Reynold Xin and Mostafa Mokhtar. 2021. Databricks Sets Official Data Warehousing Performance Record. https://databricks.com/blog/2021/11/02/databricks-sets-official-data-warehousing-performance-record.html.

[54] Reynold Xin and Josh Rosen. 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[55] MaryAnn Xue and Allison Wang. 2018. Faster SQL: Adaptive Query Execution in Databricks. https://databricks.com/blog/2020/10/21/faster-sql-adaptive-query-execution-in-databricks.html.

[56] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.. In *OSDI*, Vol. 8. 1–14.

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[58] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (October 2016), 56–65. https://doi.org/10.1145/2934664

[59] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 145–156.