Databricks
Spark SQL / Photon

Andy Pavlo // 15-721 // Spring 2023
BigQuery does **support** multi-statement transactions.
→ **INSERT/UPDATE/DELETE** queries.
→ **DDL** operations on temp tables.
→ Provides snapshot isolation via OCC.

Hadoop does **not** write map tasks shuffle output to HDFS.
→ Shuffle data is written to local disk on the data node.
ADVENT OF SPARK

High-performance and more expressive replacement for Hadoop from Berkeley.
→ Separate compute / storage
→ Support for iterative algorithms that make multiple passes on the same data set.

Written in Scala (the hot language in 2010), meaning that it ran on the JVM.

Originally only supported a low-level RDD API.
Added DataFrame API for higher-level abstraction.
**SHARK (2013)**

Modified version of Facebook's Hive middleware that converted SQL into Spark API programs.

Only supported SQL on data files registered in Hive's catalog. Spark programs could not execute SQL in between API calls.

Shark relied on the Hive query optimizer that was designed for running map-reduce jobs on Hadoop.

→ Spark has a more feature-rich native API.
Row-based SQL engine natively inside of the Spark runtime with Scala-based query codegen.

- In-memory columnar representation for intermediate results as raw byte buffers.
- Dictionary encoding, RLE, bitpacking compressions.
- In-memory shuffle between query stages.

DBMS converts a query's \textbf{WHERE} clause expression trees into Scala ASTs. It then compiles these ASTs to generate JVM bytecode.
SPARK SQL (2015)

Row-based SQL engine natively inside of the Spark runtime with Scala-based query codegen.

- In-memory columnar representation for intermediate results as raw byte buffers.
- Dictionary encoding, RLE, bitpacking compressions.
- In-memory shuffle between query stages.

DBMS converts a query’s WHERE clause expression trees into Scala ASTs. It then compiles these ASTs to generate JVM bytecode.

**Memory-based Shuffle:** Both Spark and Hadoop write map output files to disk, hoping that they will remain in the OS buffer cache when reduce tasks fetch them. In practice, we have found that the extra system calls and file system journaling adds significant overhead. In addition, the inability to control when buffer caches are flushed leads to variability in shuffle tasks. A query’s response time is determined by the last task to finish, and thus the increasing variability leads to long-tail latency, which significantly hurts shuffle performance. We modified the shuffle phase to materialize map outputs in memory, with the option to spill them to disk.
JVM PROBLEMS

Databricks' workloads were becoming CPU bound.
→ Fewer disk stalls because of NVMe SSD caching and adaptive shuffling.
→ Better filtering to skip reading data

They found it difficult to optimize their JVM-based Spark SQL execution engine further:
→ GC slowdown for heaps larger than 64GB
→ JIT codegen limitations for large methods
**DATABRICKS PHOTON (2022)**

Single-threaded C++ execution engine embedded into Databricks Runtime (DBR) via **JNI**.

→ Overrides existing engine when appropriate.

→ Support both Spark's earlier SQL engine and Spark's DataFrame API.

→ Seamlessly handle impedance mismatch between row-oriented DBR and column-oriented Photon.

Accelerate execution of query plans over "raw / uncurated" files in a data lake.
Photon: A Fast Query Engine for Lakehouse Systems

Alexander Behn, 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 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
DATABRICKS PHOTON

- Shared-Disk / Disaggregated Storage
- **Pull-based Vectorized Query Processing**
- Precompiled Primitives + Expression Fusion
- Shuffle-based Distributed Query Execution
- Sort-Merge + Hash Joins
- Unified Query Optimizer + Adaptive Optimizations
SPARK: QUERY EXECUTION

Distributed File System

```sql
SELECT language, MAX(views)
FROM wikipedia
WHERE title LIKE "%Pavlo%"
GROUP BY 1
ORDER BY 2 DESC
LIMIT 100
```
Stage #1: Partial Group By

Stage #2: Group By, Sort, Limit

Stage #3: Sort, Limit

Distributed File System

Local In-Memory Shuffle Store

SELECT language, MAX(views) FROM wikipedia WHERE title LIKE "%Pavlo%" GROUP BY 1 ORDER BY 2 DESC LIMIT 100

SPARK: QUERY EXECUTION
PHOTON: VECTORIZED QUERY PROCESSING

Photon is a pull-based vectorized engine that uses precompiled primitives for operator kernels.
→ Converts physical plan into a list of pointers to functions that perform low-level operations on column batches.

Databricks: It is easier to build/maintain a vectorized engine than a JIT engine.
→ Engineers spend more time creating specialized codepaths to get closer to JIT performance.
→ With codegen, engineers write tooling and observability hooks instead of writing the engine.
Each \textbf{GetNext} invocation on a Photon operator produces a \textit{column batch}.

→ One or more \textit{column vectors} with a \textit{position list} vector.

→ Each column vector includes a null bitmap.

Databricks: Position list vectors performs better than "active row" bitmap despite indirection.
Each **GetNext** invocation on a Photon operator produces a **column batch**.

→ One or more **column vectors** with a **position list vector**.

→ Each column vector includes a null bitmap.

**Databricks**: Position list vectors performs better than "active row" bitmap despite indirection.
Each **GetNext** invocation on a Photon operator produces a **column batch**.

→ One or more **column vectors** with a **position list vector**.

→ Each column vector includes a null bitmap.

**Databricks:** Position list vectors performs better than "active row" bitmap despite indirection.
Each \textbf{GetNext} invocation on a Photon operator produces a \textit{column batch}.

\begin{itemize}
  \item One or more \textit{column vectors} with a \textit{position list vector}.
  \item Each column vector includes a null bitmap.
\end{itemize}

Databricks: Position list vectors performs better than "active row" bitmap.

\begin{verbatim}
<table>
<thead>
<tr>
<th>col0: int32</th>
<th>col1: varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>null?</td>
</tr>
<tr>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>??</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>null?</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bbb</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>??</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>XXX</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}

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 $O(\text{batch size})$ elements instead of $O(\text{active rows})$ elements. Recent work confirms our conclusions [42].
Filter Representation in Vectorized Query Execution

Asatryan Alyosha, Traubenthal Thomas, Eremenko Oleksandr

MIT, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

A filter is a stream of binary values that is produced by a data processing pipeline. In the context of a vectorized query execution system, a filter is represented by a series of boolean values for each input tuple. This representation is used to filter out tuples that do not satisfy a given predicate, allowing for efficient data processing.

Filter Representation

Each GetNext invocation on a Photon operator produces a column batch. A column batch consists of one or more column vectors, each containing a null bitmap. Each column vector includes a position list vector. Databricks: Position list vectors outperform an active row bitmap despite indirection.

1. Introduction

Modern DBMSs utilize the vector and streaming model proposed by Vertex [7] to improve query execution performance. In this model, relations are stored as streams in memory, and operations are performed on these streams to produce new streams that are passed to downstream operators. This approach allows for efficient data processing and can significantly reduce the amount of data that needs to be stored in memory.

Vectorization and Stream Processing

Vectorization is achieved by processing large batches of data in parallel, allowing for efficient utilization of CPU resources. Stream processing, on the other hand, involves processing data as it arrives, allowing for real-time processing and analysis.

In this work, we focus on the representation of filters in a vectorized query execution system. We compare the performance of different filter representations and evaluate their effectiveness in improving query execution performance.

Figure 1: Filter Representation

- **Data**: The data column contains the actual values that are being processed.
- **Null**: The null column indicates whether a value is null or not.
- **Offset**: The offset column indicates the position of each value in the data column.
Photon does **not** support HyPer-style operator fusion so that the DBMS can collect metrics per operator to help users understand query behavior. → Vertical fusion over multiple operators in a pipeline.

Instead, Photon's engineers fuse expression primitives to avoid excessive function calls. → Horizontal fusion within a single operator.
SELECT * 
FROM A, C, 
(SELECT B.id, COUNT(*) 
FROM B 
WHERE B.val = ? + 1 
GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id
SELECT * FROM foo
WHERE cdate BETWEEN '2023-01-01' AND '2023-04-01';
VECTORWISE: PRECOMPILED PRIMITIVES

SELECT * FROM xxx
WHERE cdate >= '2023-01-01'
AND cdate <= '2023-04-01';
**VECTORWISE: PRECOMPILED PRIMITIVES**

```sql
SELECT * FROM xxx
WHERE cdate >= '2023-01-01'
  AND cdate <= '2023-04-01';
```

```c
vec<offset> sel_geq_date(vec<date> batch, date val) {
    vec<offset> positions;
    for (offset i = 0; i < batch.size(); i++)
        if (batch[i] >= val) positions.append(i);
    return (positions);
}
```

```c
vec<offset> sel_leq_date(vec<date> batch, date val) {
    vec<offset> positions;
    for (offset i = 0; i < batch.size(); i++)
        if (batch[i] <= val) positions.append(i);
    return (positions);
}
```
### Vectorwise: Precompiled Primitives

**SQL Query:**

```sql
SELECT * FROM xxx
WHERE cdate >= '2023-01-01'
AND cdate <= '2023-04-01';
```

**Vectorwise Implementation:**

```c
vec<offset> sel_between_dates(vec<date> batch,
                                date low, date high) {
    vec<offset> positions;
    for (offset i = 0; i < batch.size(); i++)
        if (batch[i] >= low && batch[i] <= high)
            positions.append(i);
    return (positions);
}
```
MEMORY MANAGEMENT

All memory allocations go to memory pool managed by the DBR in the JVM.
→ Single source of truth for runtime memory usage.

Because there are no data statistics, the DBMS has to be more dynamic in its memory allocations.
→ Instead of operators spilling its own memory to disk when it runs out of space, operators request for more memory from the manager who then decides what operators to release memory.
→ Simple heuristic that releases memory from the operator that has the least allocated but enough to satisfy request.
CATALYST QUERY OPTIMIZER

Cascades-style query optimizer for Spark SQL written in Scala that executes transformations in pre-defined stages similar to Microsoft SQL Server.

Three type of transformations:

→ **Logical**→**Logical** ("Analysis & Optimization Rules")
→ **Logical**→**Physical** ("Strategies")
→ **Physical**→**Physical** ("Preparation Rules")

Source: Cheng Lian
PHOTON: PHYSICAL PLAN TRANSFORMATION

Traverse the original query plan bottoms-up to convert it into a new Photon-specific physical plan. → New Goal: Limit the number of runtime switches between old engine and new engine.

Original Plan

- Output
- Shuffle
- Filter
- File Scan

New Plan

- Output
- Transition
- PhotonShuffle
- PhotonFilter
- File Scan
- Adapter

Source: Alex Behm
RUNTIME ADAPTIVITY

Query-Level Adaptivity (Macro)
→ Re-evaluate query plan decisions at the end of each shuffle stage.
→ Similar to the Dremel approach we discussed last class.
→ This is provided by DBR wrapper.

Batch-Level Adaptivity (Micro)
→ Specialized code paths inside of an operator to handle the contents of a single tuple batch.
→ This is done by Photon during query execution.
SPARK: DYNAMIC QUERY OPTIMIZATION

Spark changes the query plan before a stage starts based on observations from the preceding stage.
→ Avoids the problem of optimizer making decisions with inaccurate (or non-existing) data statistics.

Optimization Examples:
→ Dynamically switch between shuffle vs. broadcast join.
→ Dynamically coalesce partitions
→ Dynamically optimize skewed joins

Source: Maryann Xue
Spark (over-)allocates a large number of shuffle partitions for each stage. → Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines partitions that are underutilized using heuristics.
Spark (over-)allocates a large number of shuffle partitions for each stage. → Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines partitions that are underutilized using heuristics.

Source: Maryann Xue
Spark (over-)allocates a large number of shuffle partitions for each stage. → Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines partitions that are underutilized using heuristics.

Source: Maryann Xue
PHOTON: BATCH-LEVEL ADAPTIVITY

Separate primitives for ASCII vs. UTF-8 data
→ ASCII encoded data is always 1-byte characters, whereas UTF-8 data could use 1 to 4-byte characters.

No NULL values in a column vector
→ Elide branching to checking null vector

No inactive rows in column batch
→ Elide indirect lookups in position lists
TPC-H COMPARISON

Databricks 8 nodes + 1 driver
Scale Factor = 3000
Databricks announced audited TPC-DS results in late 2021.
Databricks announced audited TPC-DS results in late 2021.
Databricks announced audited TPC-DS results in late 2021.

"At the enterprise level, maybe some CIO is going to care about what your official TPC ranking is, but they don't make sales that way," said Carnegie Mellon University associate professor Andy Pavlo.

Databricks is gunning for Snowflake's core business

Databricks is poised to overtake Snowflake, according to a new study which shows that Databricks' systems outperformed Snowflake's data warehouse product has achieved record performance levels.

By: Williams | November 2, 2023

The rivalry between Databricks and Snowflake is about to become even more intense. And the outcome could have monumental implications for the future of the market.
The lack of statistics makes query optimization harder for queries on data lakes. Adaptivity helps for some things, but the DBMS can always do a better job if it knows something about the data.

What if there was a storage service for data lakes that supported incremental changes so that the DBMS could compute statistics?
DELTA LAKE (2019)

Transactional CRUD interface for incremental data ingestion of structured data on top of object stores.

DBMS appends writes to a JSON-oriented log. Background worker periodically convert log into Parquet files (with computed statistics).
KUDU (2015)

- Storage engine for low-latency random access on structured data files in distributed file system.
  → Started at Cloudera in 2015 to complement Impala.

- No SQL interface (must use Impala). Only supports low-level CRUD operations.
PARTING THOUGHTS

The interesting parts of Photon is in it use of precompiled primitives and its integration with an existing JVM-based runtime infrastructure.

Andy does not recommend building a Java OLAP engine from scratch.
NEXT CLASS

Snowflake