Lecture #20

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Databricks Spark SQL / Photon

Andy Pavlo // 15-721 // Spring 2023

CORRECTIONS

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.

BigQuery > Documentation > Guides

Was this helpful?

Multi-statement transactions

Send feedback

BigQuery supports multi-statement transactions inside a single query, or across multiple queries when using sessions. A multi-statement transaction lets you perform mutating operations, such as inserting or deleting rows on one or more tables, and either commit or roll back the changes atomically.

Uses for multi-statement transactions include:

- Performing DML mutations on multiple tables as a single transaction. The tables can span multiple datasets or projects.
- Performing mutations on a single table in several stages, based on intermediate computations.

Transactions guarantee ACID properties and support snapshot isolation. During a transaction, all reads return a consistent snapshot of the tables referenced in the transaction. If a statement in a transaction modifies a table, the changes are visible to subsequent statements within the same transaction.

Note: Reads from external data sources are not guaranteed to be consistent within a transaction if the underlying data source changes during the transaction.

Transaction scope

A transaction must be contained in a single SQL query, except when in `Session mode`. A query can contain multiple transactions, but they cannot be nested. You can run multi-statement transactions over multiple queries in a session.

To start a transaction, use the `BEGIN TRANSACTION` statement. The transaction ends when any of the following occur:

- The query executes a `COMMIT TRANSACTION` statement. This statement atomically commits all changes made inside the transaction.
- The query executes a `ROLLBACK TRANSACTION` statement. This statement abandons all changes made inside the transaction.
- The query ends before reaching either of these two statements. In that case, BigQuery automatically rolls back the transaction.

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.

SHARK: SQL AND RICH ANALYTICS AT SCALE
SIGMOD 2013

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.

SPARK SQL: RELATIONAL DATA PROCESSING IN SPARK
SIGMOD 2015

SPARK SQL (2015)

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

→ In-memory columnar repr

results as raw byte buffers.

→ Dictionary encoding, RLE

→ In-memory shuffle betwee

DBMS converts a query'

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.

SPARK SQL: RELATIONAL DATA PROCESSING IN SPARK
SIGMOD 2015

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
SIGMOD 2022

DATABRICKS PHOTON (2022)

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

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

PHOTON: A FAST QUERY ENGINE FOR LAKEHOUSE SYSTEMS
SIGMOD 2022

DATABRICKS PHOTON

Shared-Disk / Disaggregated Storage

Pull-based Vectorized Query Processing

Precompiled Primitives + Expression Fusion

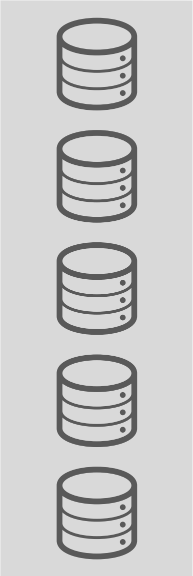
Shuffle-based Distributed Query Execution

Sort-Merge + Hash Joins

Unified Query Optimizer + Adaptive Optimizations

SPARK: QUERY EXECUTION

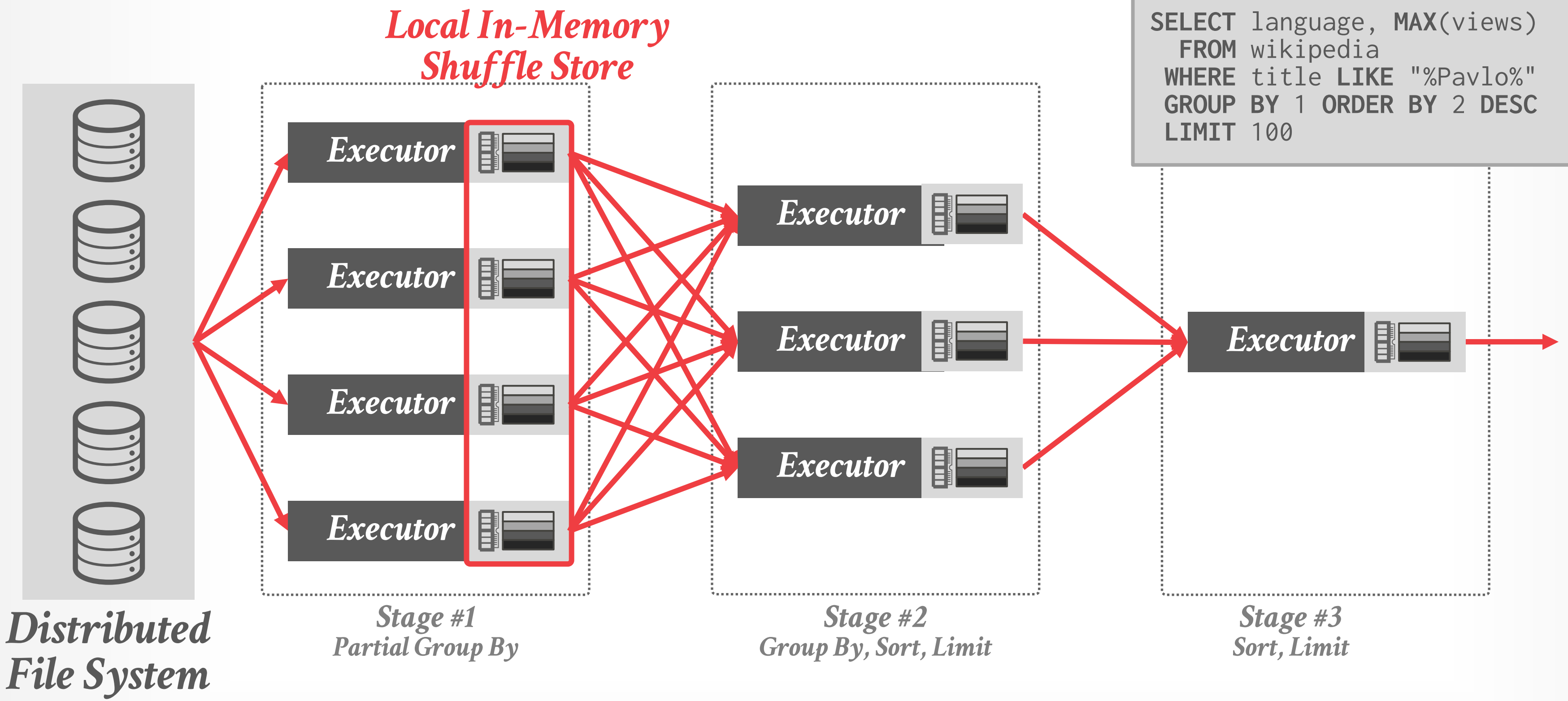
```sql
SELECT language, MAX(views)
  FROM wikipedia
 WHERE title LIKE "%Pavlo%"
 GROUP BY 1 ORDER BY 2 DESC
 LIMIT 100
```



Distributed File System

SPARK: QUERY EXECUTION

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



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.

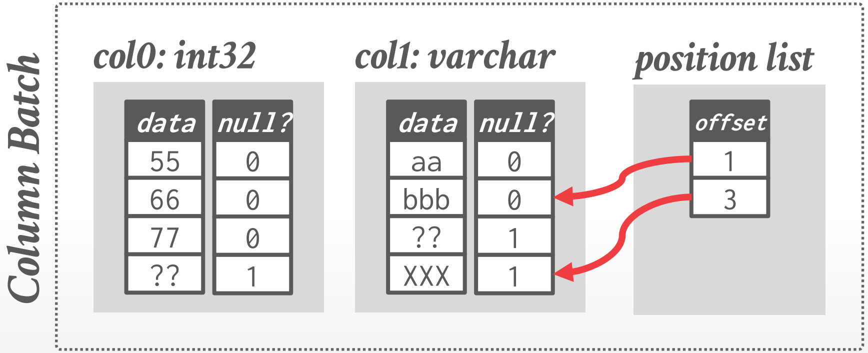
→ Engineers spend more time creating specialized codepaths to get closer to JIT performance.

→ With codegen, engineers write tooling and observability hooks instead of writing the engine.

PHOTON: VECTORIZED QUERY PROCESSING

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.



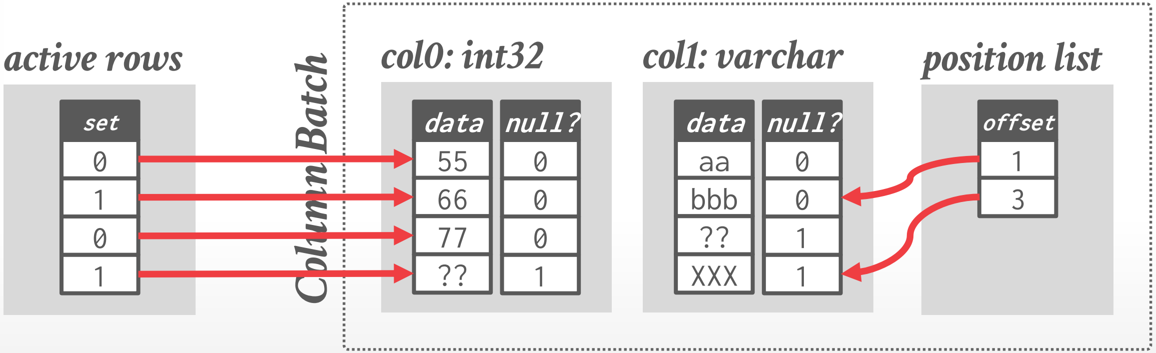
PHOTON: VECTORIZED QUERY PROCESSING

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.

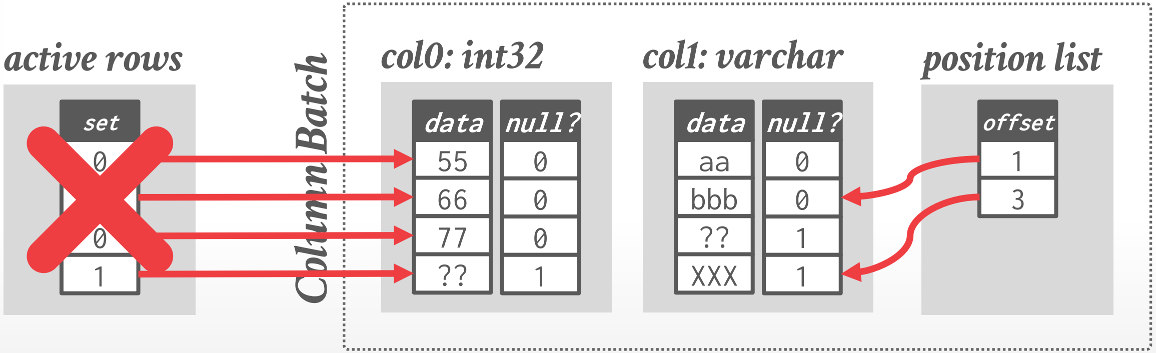
Databricks: Position list vectors performs better than "active row" bitmap despite indirection.



PHOTON: VECTORIZED QUERY PROCESSING

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.



PHOTON: VECTORIZED QUERY PROCESSING

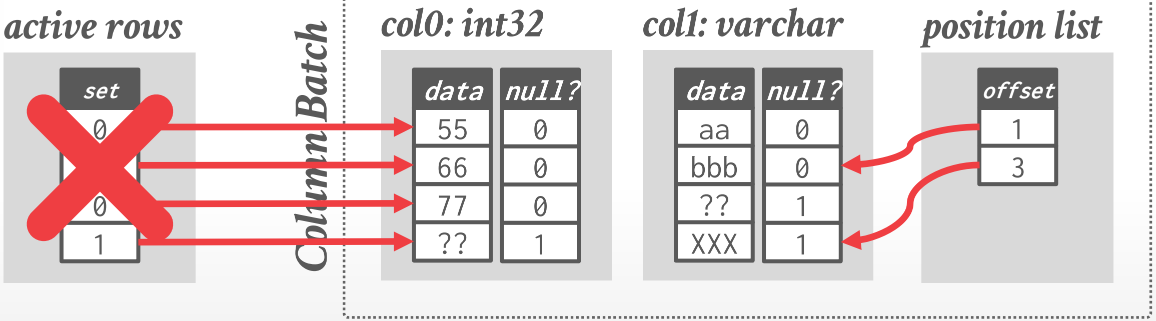
Each **GetNext** invocation on a Photon operator produces a column batch.

→ One or more colum… …ition list vector.

→ Each column vector…

Databricks: Positio… than "active row" b…

> 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].



ZED QUERY PROCESSING

ion on a Photon operator

ch.

position list vector.

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

Filter Representation in Vectorized Query Execution

Amadou Ngom[♣], Prashanth Menon
Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, Andrew Pavlo
[♣]Massachusetts Institute of Technology, Carnegie Mellon University
{ngom@mit.edu,pmenon@cs.cmu.edu}

Abstract

Advances in memory technology have made it feasible for database management systems (DBMS) to store their working data set in main memory. This trend shifts the bottleneck for query execution from disk accesses to CPU efficiency. One technique to improve CPU efficiency is batch-oriented processing, or vectorization, as it reduces interpretation overhead. For each vector (batch) of tuples, the DBMS must track the set of valid (visible) tuples that survive all previous processing steps. To that end, existing systems employ one of two data structures, or filter representations: selection vectors or bitmaps. In this work, we analyze each approach's strengths and weaknesses and offer recommendations on how to implement vectorized operations. Through a wide range of micro-benchmarks, we determine that the optimal strategy is a function of many factors: the cost of iterating through tuples, the cost of the operation itself, and how amenable it is to SIMD vectorization. Our analysis shows that bitmaps perform better for operations that can be vectorized using SIMD instructions and that selection vectors perform better on all other operations due to cheaper iteration logic.

ACM Reference Format:
Amadou Ngom[♣], Prashanth Menon and Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, Andrew Pavlo . 2021. Filter Representation in Vectorized Query Execution. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465998.3466009

1 Introduction

Modern DBMSs utilize the vectorized processing model pioneered by Vectorwise [17] to improve query execution performance. In this model, relational operators implement a uniform interface to iterate over its results in a Volcano-style manner [3]. However, unlike the original Volcano model, in a vectorized engine, relational operators exchange small vectors of typically 1–2k tuples in each invocation of the iterator. This simple enhancement (1) amortizes the iteration overhead across all tuples in the vector and (2) maximizes computation on tuple data while it is in the CPU's cache.

Vectorized relational operators exchange batches of tuple where each tuple attribute is stored separately in a compact vector. For instance, a filter operator applies a predicate on each input tuple and copies its attributes into an output vector if successful. This

DAMON'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8556-5/21/06.
https://doi.org/10.1145/3465998.3466009

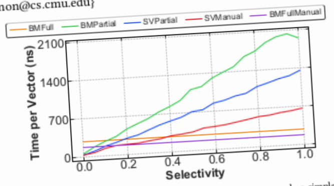


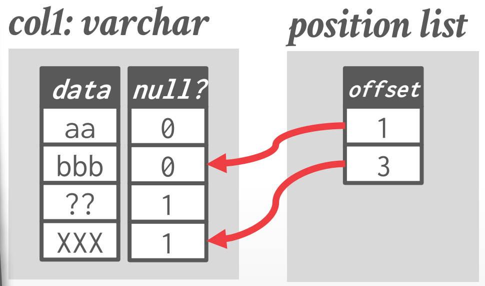
Figure 1: Motivating Example – We evaluate the time to apply a simple predicate filtering an arithmetic column with a constant value.

approach incurs memory overhead due to data copying. A common technique to overcome this is to augment batches with a data structure that *logically* masks out invalid tuples (i.e., a logical filter). We refer to this data structure as a *filter representation*. Two common representations are (1) Selection Vectors (SVs) and (2) Bitmaps (BMs). A SV is a dense sorted list of tuple identifiers (TID) indicating which tuples in the batch are valid during processing. With BMs, each tuple in the batch is assigned a positionally aligned bit; valid tuples have their bit set to 1. The DBMS marks tuples as invalid by modifying the filter representation alone without copying data.

Interestingly, previous works choose a representation strategy without providing a clear (or empirical) justification. Vectorwise and its derivatives rely selection vectors [6, 14, 15, 17]. IBM DB2's BLU [12] and the more recent VIP [11] rely on bitmaps for the intermediary results of a table scan's filters and selection vectors for other relational operators. In this work, we find that supporting both representations and dynamically choosing between them results in better performance than static implementations. Depending on the specific primitive and the selectivity (i.e., the ratio of selected tuples) of its input vector, selection vectors can outperform bitmaps and vice-versa.

To illustrate the need for a deeper exploration of the impact of a chosen filter representation strategy, we present an experiment that measures the performance of evaluating a WHERE during a sequential table scan over a table composed of a single 64-bit integer column. For this experiment, we generate the column's data using a uniform distribution, and vary the input filter's selectivity between 0 and 1. We defer the full description of our experimental setup to Section 3.

We implement and measure five different execution strategies. BMPartial, BMFull, and BMFullManual all use bitmaps. BMPartial applies the operation only on selected tuples, while BMFull applies it on *all* tuples. Likewise, BMFullManual uses a hand-written SIMD kernel to apply the operation to all tuples in each vector. SVPartial



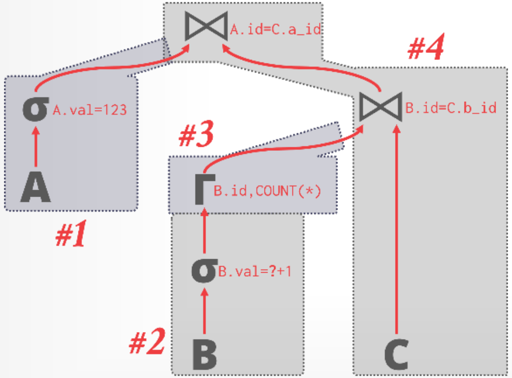
PHOTON: VECTORIZED QUERY PROCESSING

Photon does <u>not</u> 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.

HYPER: OPERATOR FUSION

```
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
```



Generated Query Plan

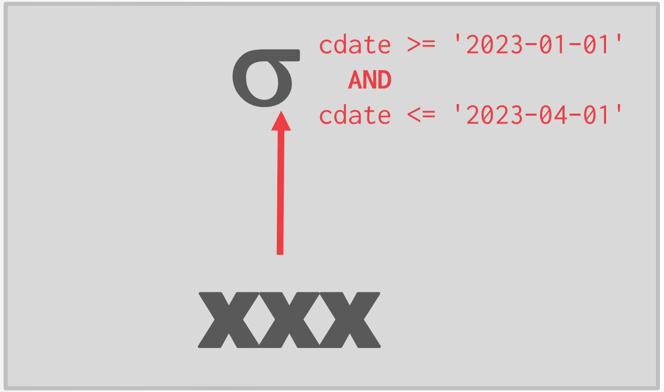
```
#1  for t in A:
      if t.val == 123:
        Materialize t in HashTable ⋈(A.id=C.a_id)

#2  for t in B:
      if t.val == <param> + 1:
        Aggregate t in HashTable Γ(B.id)

#3  for t in Γ(B.id):
      Materialize t in HashTable ⋈(B.id=C.b_id)

#4  for t3 in C:
      for t2 in ⋈(B.id=C.b_id):
        for t1 in ⋈(A.id=C.a_id):
          emit(t1⋈t2⋈t3)
```

VECTORWISE: PRECOMPILED PRIMITIVES

```
SELECT * FROM foo
 WHERE cdate BETWEEN '2023-01-01' AND '2023-04-01';
```

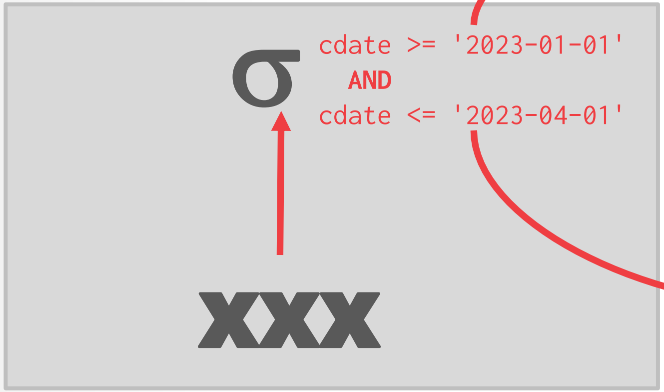
VECTORWISE: PRECOMPILED PRIMITIVES

```
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';
```

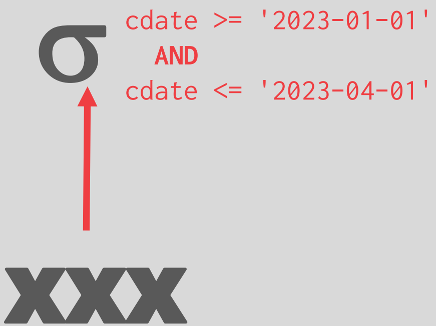



```
vec<offset> sel_geq_date(vec<date> batch, date val) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= val) positions.append(i);
  return (positions);
}
```

```
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

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




```
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.

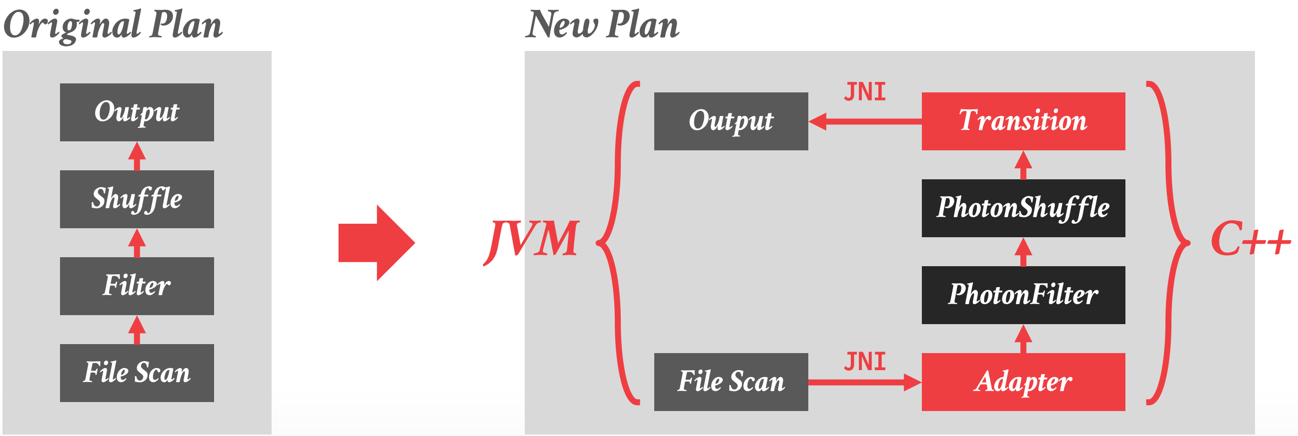
Three type of transformations:
→ **Logical→Logical** ("Analysis & Optimization Rules")
→ **Logical→Physical** ("Strategies")
→ **Physical→Physical** ("Preparation Rules")

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.



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 stages 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

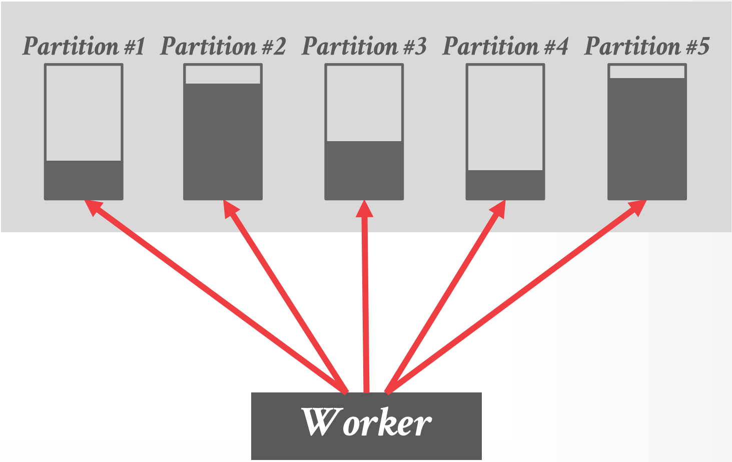
Source: Maryann Xue

SPARK: PARTITION COALESCING

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.



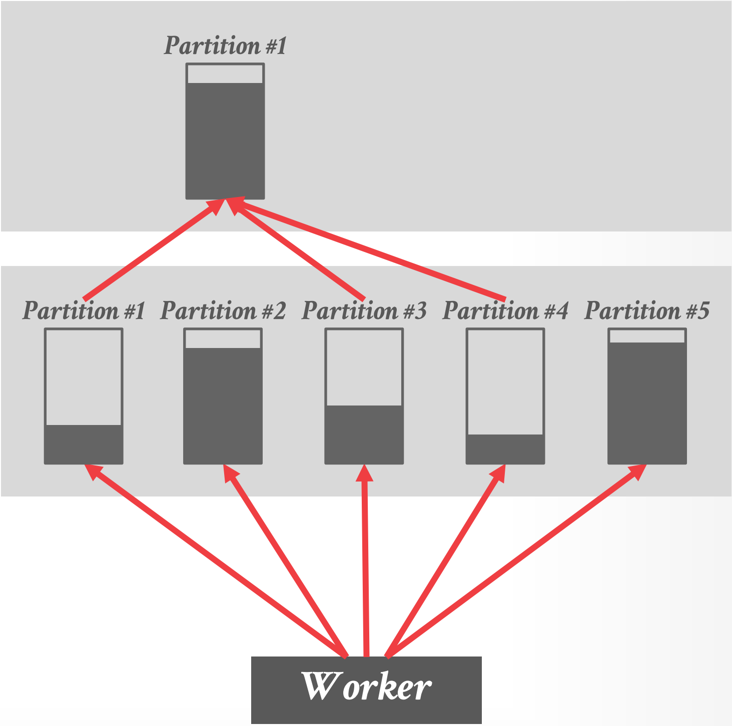
Source: Maryann Xue

SPARK: PARTITION COALESCING

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.



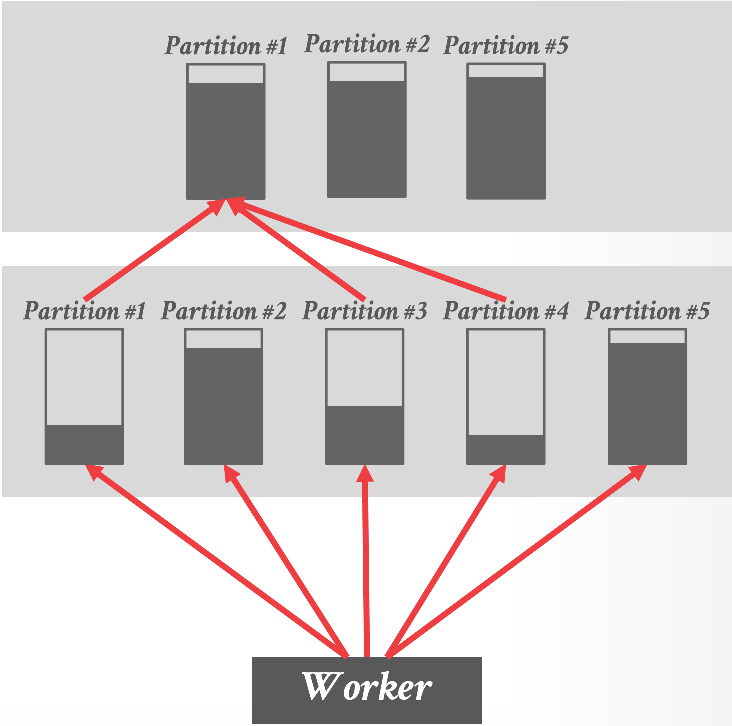
Source: Maryann Xue

SPARK: PARTITION COALESCING

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

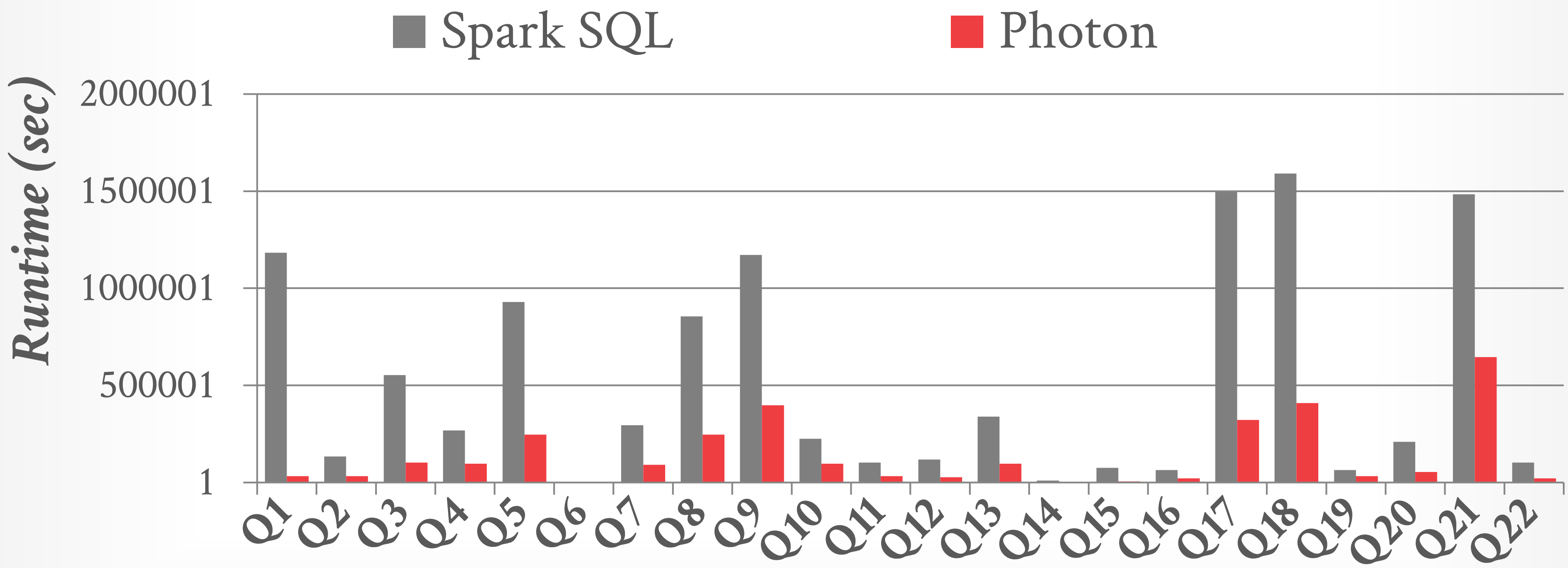
→ Elide branching to checking null vector

No inactive rows in column batch

→ Elide indirect lookups in position lists

TPC-H COMPARISON

Databricks 8 nodes + 1 driver
Scale Factor = 3000



DATABRICKS TPC-DS (2021)

Databricks announced audited TPC-DS results in late 2021.

DATABRICKS TPC-DS (2021)

Home | About the TPC | Benchmarks/Results | Downloads | TPCTC | Contact | Miscellaneous | Search | Member Login

TPC-DS V3 All Results - Sorted by Performance

Version 3 Results As of 12-Apr-2023 at 3:50 PM [GMT]

Note 1: TPC-DS Version 2 and TPC-DS Version 3 are **NOT** comparable.
Note 2: The TPC believes it is **NOT** valid to compare prices or price/performance of results in different currencies.
Note 3: The TPC believes that comparisons of TPC-DS results measured against different database sizes are misleading and discourages such comparisons. The TPC-DS results shown below are grouped by database size to emphasize that only results within each group are comparable.

10,000 GB Results

Company	System	Performance (QphDS)	Price/kQphDS	Watts/KQphDS	System Availability	Database	Operating System	Date Submitted	Cluster
Alibaba.com	Alibaba Cloud AnalyticDB	18,998,559	59.27 CNY	NR	06/17/20	Alibaba Cloud AnalyticDB 3.0.12	Alibaba Group Enterprise Linux Server 7.2 (Paladin)	06/17/20	Y
Alibaba.com	Alibaba Cloud E-MapReduce	11,569,838	237.03 CNY	NR	04/17/20	Alibaba Cloud E-MapReduce 4.0.1	CentOS Linux Release 7.4	04/16/20	Y
H3C	H3C UniServer R4900 G3	8,944,478	423.13 CNY	NR	12/23/20	GBase 8a V9	Red Hat Enterprise Linux Server 7.8	12/23/20	Y
Supermicro	Supermicro A+ Server 2123BT-HNC0R	4,418,054	110.29 USD	NR	08/31/19	Transwarp ArgoDB V1.2.1	Red Hat Enterprise Linux Server 7.6	08/07/19	Y

100,000 GB Results

Company	System	Performance (QphDS)	Price/kQphDS	Watts/KQphDS	System Availability	Database	Operating System	Date Submitted	Cluster
databricks	Databricks SQL 8.3	32,941,245	157.57 USD	NR	11/02/21	Databricks Photon Engine 8.3	Ubuntu 18.04.5 LTS	11/02/21	Y
Alibaba.com	Alibaba Cloud E-MapReduce	14,861,137	175.23 USD	NR	09/16/19	Alibaba Cloud E-MapReduce 3.21.2	CentOS Linux Release 7.4	09/16/19	Y

'NR' in the Watts/KQphDS column indicates that no energy data was reported for that benchmark.

DATABRICKS

TPC®

Home | About the TPC | Benchmarks/Results | Downloads | TPCTC

TPC-DS V3 All Results - Sorted by Performance

Version 3 Results As of 12-Apr-2023 at 3:50 PM [GMT]

Note 1: TPC-DS Version 2 and TPC-DS Version 3 are **NOT** comparable.
Note 2: The TPC believes it is **NOT** valid to compare prices or price/performance of results in differe
Note 3: The TPC believes that comparisons of TPC-DS results measured against different databas

Company	System	Performance (QphDS)	Price/kQphDS
	G3		
Supermicro	Supermicro A+ Server 2123BT-HNC0R	4,418,054	110.29 USD

100,000 GB Results

Company	System	Performance (QphDS)	Price/kQphDS
databricks	Databricks SQL 8.3	32,941,245	157.57 USD
Alibaba.com	Alibaba Cloud E-MapReduce	14,861,137	175.23 USD

'NR' in the Watts/KQphDS column indicates that no energy data was reported for that benchma

"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.

protocol

SUBSCRIBE

ENTERPRISE

Databricks is gunning for Snowflake's core business

In a shot across the bow to Snowflake, Databricks is set to announce on Tuesday that its flagship data warehouse product has achieved record performance levels.

Databricks is poised to announce that an independent industry group validated results which show that Databricks' systems outperformed the closest data warehouse competitor by 2.2x. | Photo: Databricks

By Joe Williams | November 2, 2021

Most Popular

The rivalry between Databricks and Snowflake is about to become even more hostile. And the outcome could have monumental ramifications for one of the most found...

Bulletins

OBSERVATION

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).

DELTA LAKE: HIGH-PERFORMANCE ACID TABLE STORAGE OVER CLOUD OBJECT STORES
VLDB 2020

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.

KUDU: STORAGE FOR FAST ANALYTICS ON FAST DATA
WHITE PAPER 2015

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