Lecture #18: System Analysis (Databricks / Spark)

15-721 Advanced Database Systems (Spring 2024)

https://15721.courses.cs.cmu.edu/spring2024/

Carnegie Mellon University
Prof. Andy Pavlo

1 Background of Photon

This section offers a comprehensive summary of the prior advancements and accompanying challenges that paved the way for the emergence of Photon.

1.1 Advent Of Spark

Spark is a high-performance and more expressive replacement for Hadoop from Berkeley. It separates compute and storage as it has HDFS and extra execution nodes like Hadoop, but it also supports iterative algorithms that make multiple passes on the same dataset, which you couldn't do with Hadoop easily.

Spark is written in Scala, meaning that it runs on the JVM. Its first version only supports a low-level RDD API, but later versions introduce DataFrame API for higher-level abstraction.

1.2 Shark (2013)

[5] Shark modifies the version of Facebook's Hive middleware that converts SQL into Spark API programs.

Its limitation is that it only supports SQL on data files registered in Hive's catalog. Also, SQL queries and API calls cannot be mixed since Spark programs could not execute SQL in between API calls.

Another challenge it faces is that Shark relies on the Hive query optimizer that is designed for running map-reduced jobs on Hadoop, so the generated Spark program is not as efficient as the one written by hand because Hive optimizer isn't aware that Spark has a more feature-rich native API.

1.3 Spark SQL (2015)

[1] Spark SQL is a row-based SQL engine natively inside the Spark runtime with Scala-based query code generation. It uses in-memory columnar representation for intermediate results as raw byte buffers. It also uses dictionary encoding, RLE, bitpacking compression methods.

For its in-memory shuffle phase between query stages, originally it just relies on the OS page cache to keep data in memory, and then spills them to disk when necessary, but involving OS may not be a good idea.

Also, DBMS converts a query's **WHERE** clause expression trees into Scala ASTs. It then compiles these ASTs to generate JVM bytecode, but there is too much computational overhead, making SQL queries CPU-bound rather than disk-bound.

1.4 JVM Problems

However, there are some limitations with JVM-based Spark SQL execution engine:

1. Databricks' workloads were becoming CPU-bound at that time:
   - Fewer disk stalls because of NVMe SSD caching and adaptive shuffling
   - Better filtering to skip reading data
2. It was difficult to optimize their JVM-based Spark SQL execution engine further:
   - JVM's GC slowdown for heaps larger than 64GB
   - JIT code generation limitations for larger methods

2 Databricks Photon (2022)

[3] Databricks Photon is a single-threaded C++ execution engine embedded into Databricks Runtime (DBR) via JNI (Java Native Interface).

1. It overrides existing engine when suitable and necessary without changing the high level behaviors to make this completely transparent to users.
2. It supports both Spark's earlier SQL engine and Spark's DataFrame API.
3. It seamlessly handles impedance mismatch between row-oriented DBR and column-oriented Photon.

In short, some portion of a query will be run in Photon if there is an accelerated Photon implementation of that task or operator correspondingly.

2.1 High-Level Overview

Photon has some key designs listed below:

1. It uses shared-disk, also with disaggregated compute and storage.
2. It implements pull-based vectorized query processing.
3. It adopts precompiled primitives and expression fusion, rather than JIT code generation.
4. It uses shuffle-based distributed query execution.
5. It supports sort-merge and hash joins.
6. It supports unified query optimizer and adaptive optimizations.

2.2 Vectorized Query Processing

Photon is a pull-based vectorized engine that uses precompiled **operator kernels** (also called **primitives** for the vector-wise version), which is some precompiled code that can do small computations or tasks on a vector of data. Also, Photon converts physical plan into a list of pointers to functions that perform low-level operations on column batches. The way to decrease the cost of jumping to these different function pointers is not only handling multiple tuples within a single vector, but also combining multiple primitives if they are used continuously, which is called "expression fusion".

The reason why Photon adopts precompiled code rather than JIT code generation engine is that the former one is much easier to build and maintain. For code generation engine, engineers have to develop a lot of tooling and observability hooks for debugging purposes instead of writing the engine. So with a vectorized engine, engineers have more time to create specialized code paths and iterating to get closer to JIT performance.

All the operators support the **GetNext** function, whose invocation produces a column batch. It passes the dead tuples and uses a position list vector to track the active tuples as they move up in the query plan. Noticeably, an alternative approach is using an "active row" bitmap to indicate which tuple is active via one bit per tuple, but the position list performs better in general.

As mentioned, Photon does not use Hyper-style operator fusion so that the DBMS can collect isolated metrics per operator to help users understand query behavior. Instead, Photon fuses expression primitives to avoid excessive function calls.

2.3 Expression Fusion

If several primitives are being used one after another, over and over again within an operator, Photon can fuse them into a single primitive. It is a horizontal fusion within a single operator, not a vertical fusion over multiple operators in a pipeline like Hyper-style operator fusion.

2.4 Memory Management

All the memory allocations go to memory pool managed by the DBR in the JVM. Also, the DBMS has to be more dynamic in its memory allocations due to a lack of data statistics. Instead of every operator 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. It uses simple heuristic that releases memory from the operator that has the least allocated but enough to satisfy request.

2.5 Physical Plan Transformation

Spark SQL has a cascades-style query optimizer called Catalyst. Apart from logic-to-logic and logic-to-physical transformations, it also has a physical-to-physical transformation to determine where to inject Photon operators as needed.

For this physical plan transformation in query optimization, Catalyst traverses the original plan bottoms-up to convert it to a Photon-specific physical plan, and the new goal is to limit the number of runtime switches between old engine (Java) and new engine (C++).

2.6 Micro Runtime Adaptivity

Photon can do batch-level adaptivity, which specializes code paths inside an operator to handle contents of a single tuple batch. There are more details in the next section.

3 Runtime Adaptivity

In the Data Lake environment, we generally don't have any data statistics, so the DBMS needs to do some runtime adaptive optimizations.

3.1 Query-Level Adaptivity (Macro)

This is provided by DBR wrapper, leveraging statistics collected at the end of each shuffle stage to re-evaluate previous query plan decisions. It is similar to the Dremel approach discussed last class.

This adaptive query optimization in Spark changes the query plan before a stage starts based on observations from the preceding stage, to avoid the problem of optimizer making decisions with inaccurate (or non-existing) data statistics. For example, it can dynamically switch between shuffle and broadcast join, dynamically coalesce partitions, and dynamically optimize skewed joins.

Spark handles overflowing partitions in a slightly different way with Dremel, which is called Partition Coalescing. Actually Spark can't expand the number of partitions, so it (over-)allocates a large number of shuffle partitions for each stage, more than actually needed to avoid one partition from filling up too much. After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.

3.2 Batch-Level Adaptivity (Micro)

This is done by Photon automatically during query execution. Photon can choose the best operator implementation for a single tuple batch inside an operator based on the data it has seen. It is similar to Velox optimizations discussed in Lecture #05.

For example, it supports the following optimizations:

1. **Custom Primitives for ASCII and UTF-8 Data**: ASCII encoded data is always 1-byte characters, whereas UTF-8 data could use 1 to 4-byte characters, so it can switch to a faster implementation for ASCII data.
2. **Compact Sparse Vectors**: It can copy tuples to new vectors before probing hash tables to maximize SIMD utilization.
3. **No NULL Values in a Vector**: It can elide branching to checking null vector.

4. **No Inactive Rows in a Vector**: It can elide indirect lookups in position lists.

4 Spark Accelerators

Since Photon is proprietary, there are other open-source alternatives to accelerate Spark's runtime. These systems redirect entire query plans to separate runtime engines rather than use Photon's fine-grain integration, which means they ignore the Spark runtime entirely and run the queries on a separate engine. Here are some notable examples:

1. Apache Gluten (Intel)
2. PAPIDS Accelerator for Spark (NVIDIA)
3. Blaza (Kuaishou)
4. Datafusion Comet (Apple)

5 Storage Services for Data Lake with Statistics

The lack of statistics makes query optimization harder for queries on data lakes. Adaptivity helps for some things, but the DBMS can do a better job if it knows something about the data. So what if there is a storage service for data lakes that supports incremental changes so that the DBMS could compute statistics?

5.1 Delta Lake (2019)

[2] Delta Lake is a transactional CRUD interface for incremental data ingestion of structured data on top of object stores. DBMS appends writes to a JSON-oriented log, and the background workers periodically convert log into Parquet files, also with computed statistics.

5.2 Apache Kudu (2015)

[4] Apache Kudu is a storage engine for low-latency random access on structured data files in distributed file system. The updates are written to in-memory B+ tree and then are converted to column store when written to disk. It also collects statistics when the data is ingested, similar to Delta Lake. It only supports low-level CRUD operations and there is no SQL interface (must use Impala).

5.3 Apache Hudi (2016)

Apache Hudi is a transactional (MVCC) system for incremental data ingestion of structured data on top of object stores. It keeps track of partitioning, versioning, schema changes, and background compaction. It also provides catalog service for runtime lookups and pruning of meta-data. And it not only supports both Parquet and ORC file formats, but supports data ingestion from multiple sources like Kafka, Spark SQL, and Flink SQL.

5.4 Apache Iceberg (2017)

Apache Iceberg is an infrastructure and file format extension to Parquet for maintaining catalog about data files in an object store. It keeps track of partitioning, versioning, and schema changes. It also provides catalog service for runtime lookups and pruning of meta-data. Snowflake has added support for ingesting, creating, and querying Iceberg files.

6 Conclusion

The interesting parts of Photon are its use of precompiled primitives and its integration with an existing JVM-based runtime infrastructure, since it is not a good idea to build a Java OLAP engine from scratch.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742797. URL https://doi.org/10.1145/2723372.2742797.

[2] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. undefinedwitakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta lake: high-performance acid table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415560. URL https://doi.org/10.14778/3415478.3415560.

[3] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. van Bussel, H. van Hovell, M. Xue, R. Xin, and M. Zaharia. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2326–2339, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526054. URL https://doi.org/10.1145/3514221.3526054.

[4] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, et al. Kudu: Storage for fast analytics on fast data. *Cloudera, inc*, 28:36–77, 2015.

[5] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 13–24, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465288. URL https://doi.org/10.1145/2463676.2465288.