Lecture #05: Query Execution & Processing II

15-721 Advanced Database Systems (Spring 2024)
`https://15721.courses.cs.cmu.edu/spring2024/`
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

There are many ways to execute queries, and this lecture will go over different techniques to execute analytical workloads (in OLAP systems) in an efficient and effective manner.

The topics covered will be as follows:

- Execution Parallelism
- Execution Engines
- Execution Operator Output
- Intermediate Data Representation
- Expression Evaluation
- Adaptive Execution

2 Execution Parallelism

As a high performance system, a DMBS will want to maximally utilize the hardware it is running on. This means allowing the DBMS to execute multiple tasks concurrently and in parallel on multiple cores. Note that this is different from the ”data parallelism” concepts that were covered in the past few lectures, as execution parallelism can be more broadly defined as ”computational parallelism”.

One way we can do this is by chopping a query into multiple small pieces, or ”pipelines.” By doing this, we can have multiple threads or even multiple processes working on separate pipelines in parallel. There are generally two types of query parallelism: **inter-query parallelism** and **intra-query parallelism**. Note that these types of parallelism are *not* mutually exclusive.

2.1 Inter-Query Parallelism

Inter-query parallelism implies multiple queries running at the same time, usually on multiple nodes in a distributed system. To coordinate these queries, a common approach is to have the scheduler use a FIFO policy. Many commercial systems may adopt some sort of prioritization, but at a high level they still use the same FIFO policy.

2.2 Intra-Query Parallelism

Intra-query parallelism implies parallelism within a single query. A DBMS can achieve this by running execution operators in parallel by assigning tasks to different workers (threads, processes, or even entire nodes). For intra-query parallelism, we also have the notion of **inter-operator parallelism** and **intra-operator parallelism** (which are also not mutually exclusive).

Inter-operator parallelism implies that execution operators can run in parallel. For example, we can have two table scans running at the same time since they do not need to be synchronized with each other. Inter-operator parallelism can also be called *vertical* or even *pipelined* parallelism.

Intra-operator parallelism, which can also be called *horizontal* parallelism, involves parallelism within a specific operator. This entails applying multiple instances of the same function to disjoint sets of the input data. To facilitate this, execution engines can insert an **Exchange** operator.

The exchange operator can come in 3 forms: Gather, Distribute, and Repartition / Shuffle.

1. Gather: Exchanges data from N inputs into 1 output
2. Distribute: Exchanges data from 1 input to N outputs
3. Repartition / Shuffle: Exchanges data from M inputs to N outputs

3 Execution Engines

Usually it's a good idea to use a standalone execution engine library for vectorized execution on columnar data. For example:

1. DataFusion
2. OAP from Intel
3. Polars
4. Velox from Meta

3.1 Velox

Velox [1] is an extensible C++ library aimed at supporting high-performance, single-node query execution. This means that it does not have a SQL parser, meta-data catalog, or optimizer.

Velox processes queries through a physical plan (DAG of operators), supports vectorized and adaptive query optimization, and is compatible with Apache Arrow. Velox interfaces with storage systems like S3 and HDFS through connectors and supports formats like Parquet and ORC/DWRF. Velox's core components include a type system, function API, storage connectors / adapters, resource manager, expression engine, internal data representation, and operator engine.

4 Execution Operator Output

Execution engines must figure out what exactly their operators should output once they are finished with their computation. Usually, it is a decision between early materialization and late materialization.

Early materialization means that the operator fully creates (materializes) the physical representation in bits of the output tuples. Late materialization, on the other hand, only adds extra information gained from the operators, and does not fully materialize the tuples.

Early materialization is generally easier for row stores since you do not have to keep track of what information is attached to non-materialized tuples, but it comes at the cost of overhead for the formulation of each of the tuples. However, for column-stored data, late materialization is easier since you only need to copy columns of input data and output some extra columns of information. So late materialization is generally the better choice for OLAP systems.

5 Intermediate Data Representation

Execution engines must also figure out how to send data in between their operators. In order to send data, they must figure out how to represent their intermediate state. We call this state *intermediate representation*, and the encoding of the propagated in-memory data should be consistent throughout the DBMS.

Ideally, the intermediate representation should have two properties: the data should be movable between tasks *without* serialization and deserialization, and it should support zero-copy shared memory access (in other words, sharing data should be a simple as copying a pointer rather than deep copying all of the data).

5.1 Apache Arrow

One of the main open-source intermediate representation formats is Apache Arrow. Arrow is an in-memory language-agnostic columnar data format with high cache efficiency. Arrow supports both random and sequential data access, and it is intended for use with vectorized execution engines.

In the Arrow format, all data is fixed-size, and the only compression schemes that Arrow supports is Run-Length Encoding and Dictionary compression. For variable-length data like strings, Arrow stores offsets as pointers into a buffer.

Notably, Velox extended Arrow with German/Umbra-styled string storage. Instead of just storing an offset, they store a size (4 bytes), a prefix (4 bytes), and a payload pointer (8 bytes) to enable faster prefix lookups.

6 Expression Evaluation

Logical expressions (usually needed for predicates in a WHERE clause) are represented as an expression tree. The nodes can be different expressions:

1. Comparisons $(! =, =, <, >)$
2. Conjunctions and Disjunctions
3. Arithmetic Operators
4. Constants
5. Functions

Traversing and crawling through the expression tree is generally very bad for the CPU. The better approach is to evaluate the expression directly, or to vectorize it to evaluate a batch of tuples at the same time.

Velox uses several techniques to flatten the expression tree:

1. Constant folding: Computes a sub-expression of a constant value and then reuses the result
2. Common Sub Expression Elimination: Finds a duplicate sub-expression that can be shared across the tree
3. Velox has an experimental branch that generates C++ code to be compiled

7 Adaptive Query Processing

The execution engine is only as good as the query plans it receives. If it receives a bad query plan, all optimizations that the execution engine makes will be negated.

Adaptive Query Processing allows the execution engine to modify the query plan or expression tree while it is running its operators. It uses the information gained from execution to make decisions for later execution.

Note that only a few systems do this because this is difficult.

Velox uses a few different adaptive query processing techniques:

1. Predicate Reordering: reorder the predicate based on their selectivity and computation cost
2. Column Pre-fetching: asynchronous retrieval of columns (not pages) during expression evaluation
3. Not Null Fast Paths: If we detect that there are no NULLs in a column, then we don't need to do NULL check for every single value
4. Elide ASCII Encoding Check: If we detect that a string is not UTF-8 encoded, we can use faster ASCII functions
5. Reusing Buffers

References

[1] M. B. K. W. L. S. K. P. W. H. B. C. Pedro Pedreira, Orri Erling. Velox: Meta's unified execution engine. 2022. doi: https://dl.acm.org/doi/10.14778/3554821.3554829.