Lecture #03: Query Compilation

15-721 Advanced Database Systems (Spring 2018)

`http://15721.courses.cs.cmu.edu/spring2018/`

Carnegie Mellon University

Prof. Andy Pavlo

1 Background

After switching to an in-memory DBMS, the only ways to increase throughput is to reduce the number of instructions executed [4]:

- To go 10× faster, the DBMS must execute 90% fewer instructions.
- To go 100× faster, the DBMS must execute 99% fewer instructions.

One way to achieve such a reduction is through *code specialization*. This means generating code that is specific to a particular task in the DBMS (e.g., a specific query).

2 Query Processing

There are three ways for a DBMS to execute a query plan:

- **Tuple-at-a-time**: Each operator calls **next** on their child to get the next tuple to process. Also known as the *Volcano* [5] iterator model.
  Example: This is the approach used by most DBMSs.
- **Operator-at-a-time**: Each operator materializes their entire output for their parent operator. This approach is ideal for in-memory OLTP engines because it reduces the number of function calls and the number of tuples emitted per operator is small.
  Example: H-Store/VoltDB, MonetDB.
- **Vector-at-a-time**: Each operator calls **next** on their child to get the next **batch** of data to process.
  Example: VectorWise [2], Peloton [10].

Predicate Interpretation:

- DBMS evaluates predicates using an expression tree.
- Expression trees are expensive to interpret when a query accesses a lot of tuples.

3 Code Specialization

Any CPU intensive entity of database can be natively compiled if they have a similar execution pattern on different inputs.

- Access methods
- Stored procedure
- Operator execution
- Predicate evaluation

- Logging operations

Benefits of Code Specialization:

- Attribute types are known *a priori*; data access function calls can be converted to in-line pointer casting.
- Predicates are known *a priori*; the DBMS can evaluate them using primitive data comparisons.
- No function calls in loops; this allows the compiler to efficiently distribute data to registers and increase cache reuse.

4 Code Generation

Approach #1 – Transpilation (Source-to-Source Compilation)

Write code that converts a relational query plan into C/C++ and then run it through a conventional compiler to generate native code [8]:

- For a given query plan, generate a C/C++ program that implements that query's execution.
- Use an off-shelf compiler (e.g., `gcc`) to convert the code into a shared object, link it to the DBMS process, and invoke the exec function to execute the query.
- The generated query code can invoke any other function in the DBMS.
- This allows it to use all the same components as interpreted queries (e.g. concurrency control, logging/checkpoints).
- The evaluation of the HIQUE [8] system shows that the DBMS incurs fewer memory stalls when executing the query but the compilation time is long (i.e., greater than 100-600 ms).

Approach #2 - JIT Compilation

Generate an intermediate representation (IR) of the query that can be quickly compiled into native code [11].

- Organizes query processing in a way to keep a tuple in CPU registers for as long as possible. The query plan is divided into pipelines (i.e., how far up the query tree the DBMS can continue processing a tuple before needing the next tuple becomes necessary).
  - Push-based vs. Pull-based
  - Data-Centric vs. Operator-Centric
- The DBMS can compile queries into native code using the LLVM toolkit [9]:
  - Collection of modular and reusable compiler and tool chain technologies.
  - Core component is a low-level programming language (IR) that is similar to assembly.
  - Not all of the DBMS components need to be written in LLVM IR. The LLVM code can make calls to C++ code.
- Query Compilation Cost:
  - LLVM compilation time grows super-linearly relative to the query size (# of joins, predicates, and aggregations).
  - Not a big issues with OLTP applications. Major problem with OLAP workloads.

One solution to mask the compilation time is HyPer's *Adaptive Execution* model [6]:

1. First generate the LLVM IR for the query.
2. Execute the IR in an interpreter while compiling the query in a background thread.
3. When the compiled query is ready, seamlessly replace the interpretive execution.

5 Real World Implementations

- **IBM System R** [3]
  - A primitive form of code generation and query compilation was used by IBM in 1970s.
  - Compiled SQL statements into assembly code by selecting code templates for each operator.
  - Technique was abandoned when IBM built **DB2** in the 1980s.
- **Oracle**
  - Convert PL/SQL stored procedures into `Pro*C` code and then compiled into native C/C++ code.
  - They also put Oracle-specific operations directly in the SPARC chips as co-processors.
- **Microsoft Hekaton** [4]
  - Can compile both procedures and SQL.
  - Non-Hekaton queries can access Hekaton tables through compiled inter-operators.
  - Generates C code from an imperative syntax tree, compiles it into DDL, and links at runtime.
- **Cloudera Impala** [7]
  - LLVM JIT compilation for predicate evaluation and record parsing.
  - Optimized record parsing is important for Impala because they need to handle multiple data formats stored on HDFS.
- **Actian Vector** (formerly **VectorWise**) [13]
  - Pre-compile thousands of "primitives" that perform basic operations on typed data.
  - The DBMS then executes a query plan that invokes these primitives at runtime.
- **MemSQL** (pre-2016)
  - Performs the same C/C++ code generation as HIQUE [8] and then invokes `gcc`.
  - Converts all queries into a parameterized form and caches the compiled query plan.
- **MemSQL** (Since 2016) [12]
  - A query plan is converted into an imperative plan expressed in a high-level imperative DSL called the *MemSQL Programming Language* (MLP).
  - The DSL then gets executed into a second language of opcodes
  - Finally the DBMS compiles the opcodes into LLVM IR and then to native code.
- **VitesseDB**
  - Query accelerator for Postgres/Greenplum that uses LLVM + intra-query parallelism.
- **Apache Spark** [1]
  - Introduced in the new Tungsten engine in 2015 that included code generation.
  - The system converts a query's `WHERE` clause expression trees into an AST.
  - It then compiles these ASTs to generate JVM byte code that it executes natively.
- **Peloton** [10]
  - Full compilation of the entire query plan
  - Relax the pipeline breakers of HyPer to create mini-batches for operators that can be vectorized.
  - Use software pre-fetching to hide memory stalls.

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, pages 1383–1394, 2015. URL `http://doi.acm.org/10.1145/2723372.2742797`.

[2] P. A. Boncz, M. Zukowski, and N. Nes. In *CIDR*, pages 225–237, 2005. URL `http://cidrdb.org/cidr2005/papers/P19.pdf`.

[3] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, Oct. 1981. URL `http://doi.acm.org/10.1145/358769.358784`.

[4] C. Freedman, E. Ismert, and P.-Å. Larson. Compilation in the microsoft sql server hekaton engine. *IEEE Data Eng. Bull.*, 37:22–30, 2014. URL `http://15721.courses.cs.cmu.edu/spring2017/papers/20-compilation/freedman-ieee2014.pdf`.

[5] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994. URL `http://dx.doi.org/10.1109/69.273032`.

[6] A. Kohn, V. Leis, and T. Neumann. Generating code for holistic query evaluation. In *ICDE)*, 2018. URL `https://db.in.tum.de/~leis/papers/adaptiveexecution.pdf`.

[7] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015. URL `http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf`.

[8] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624, March 2010. URL `http://10.1109/ICDE.2010.5447892`.

[9] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

[10] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017. URL `http://www.vldb.org/pvldb/vol11/p1-menon.pdf`.

[11] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4 (9):539–550, June 2011. URL `http://dx.doi.org/10.14778/2002938.2002940`.

[12] D. Paroski. Code Generation: The Inner Sanctum of Database Performance, September 2016. URL `http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html`.

[13] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, 2013. URL `http://doi.acm.org/10.1145/2463676.2465292`.