Lecture #07: Code Generation & Compilation

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

1 Background

In the previous lecture, we focus on how to use SIMD to vectorize core database algorithms for sequential scans. The research literature in the 2010s can give the impression that vectorization and JIT compilation are mutually exclusive.

After minimizing the disk I/O during query execution, the only way 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.

2 Observation

One way to achieve a significant reduction in instructions is through **code specialization**. This means generating code (e.g., machine code and source code) that is specific to a particular task (e.g., one query) in the DBMS (e.g., a specific query) on the fly. Most code prioritizes human readability over performance, which is particularly in query interpretation and expression evaluation.

Query Interpretation:

- In a volcano model, each input tuple from the child is iterated and emitted to upper-level operators. This involves having a pointer to a root node in the query plan, and there will be virtual function table lookups during runtime in languages like C++.
- However, relying on function pointer lookups at runtime can be inefficient on modern CPUs.

Expression Evaluation:

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

3 Code Specialization

The DBMS generates code for any CPU-intensive task that has a similar execution pattern on different inputs. The operations include:

- Access methods
- Stored procedure
- Operator execution
- Predicate evaluation (Most Common)

- Logging operations

For query-focused compilation, the DBMS (typically) specializes in it after generating the physical plan for a query.

Because relational DBMSs have schemas, code specialization has three benefits:

- Since attribute types are known *a priori*, the DBMS can convert data access function calls to inline pointer casting.
- Since predicates are known *a priori*, the DBMS can evaluate them using primitive data comparisons.
- The DBMS can have no function calls in loops. This allows the compiler to efficiently distribute data to registers and increase cache reuse.

4 Code Generation

When executing a query, the DBMS first sends it to the parser to produce the abstract syntax tree (AST). The AST is then passed to the binder, which communicates with the system catalog to retrieve the annotated AST. The Annotated AST is then translated to a physical plan by the optimizer. In the end, the physical plan is sent to the compiler, which uses **Code Generation** approaches to generate native code.

There are two approaches to code generation:

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

Transpilation involves writing code that converts a relational query plan into imperative language source code and then runs it through a conventional compiler to generate native code. Relational operators are a useful way to reason about a query but are not the most efficient way to execute it.

- **Holistic Code Generation:** For a given query plan, transpilation generates a C/C++ program that implements the query's execution, baking in all the predicates and type conversions. It then uses 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.
- **DBMS Integration:** The generated query code can invoke any other function in the DMBS. This allows the generated code to use all the same components as interpreted queries (e.g., network handlers, buffer pool manager, concurrency control, logging/checkpoints, and indexes). Debugging is (relatively) easy because you step through the generated source code.
- **Observation:** Relational operators are a useful way to reason about a query but are not the most efficient way to execute it. The evaluation of the **HIQUE** [9] system shows that the DBMS incurs fewer memory stalls when executing the query, but it takes a (relatively) long time to compile a C/C++ source file into executable code. (i.e., greater than 100–600 ms). HIQUE also does not allow for full pipelining.

Approach #2 - JIT Compilation

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

- **HyPer:** The **HyPer** DBMS compiles queries into native code using the LLVM toolkit [10]. Instead of emitting C++ code, HyPer emits LLVM IR. LLVM is a collection of modular and reusable compiler and toolchain technologies. Its core component is a low-level programming language (IR) that is similar to assembly. Like transpilation, LLVM doesn't need to implement all of the DBMS components in IR. The LLVM code can make calls to C++ code.

- **JIT Query Compilation:** JIT compilation fuses operators aggressively within pipelines 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). This approach is push-based and data-centric.
- **Observation:** LLVM compilation cost is low with OLTP queries but may have major problems with OLAP workloads. HyPer's query compilation time grows super-linearly relative to the query size (# of joins, predicates, and aggregations).

One solution to mask the compilation time is **HyPer**'s **Adaptive Execution** model [7]:

1. The model generates LLVM IR for the query and immediately starts executing the IR using an interpreter.
2. The DBMS then compiles the query in the background.
3. When the compiled query is ready, seamlessly replace the interpretive execution. For each morsel, check to see whether the compiled version is available.

5 Real World Implementations

Real-world implementations of database management systems (DBMS) vary widely in their approaches and techniques. They can be classified into different categories based on their implementation strategies, including transpilation, custom solutions, JVM-based implementations, and LLVM-based implementations.

Classifications:

- **Transpilation**: Amazon Redshift, Oracle, MemSQL (2016)
- **Custom**: IBM System R, Actian Vector, Microsoft Hekaton, SQLite, TUM HyPer, TUM Umbra, QuestDB
- **JVM-based**: Spark, Neo4j, Splice Machine, Presto / Trino, OrientDB, Tajo, Derby
- **LLVM-based**: SingleStore, VitesseDB, PostgreSQL (2018), CMU Peloton, CMU NoisePage, TUM LingoDB

Below, we explore notable examples of these implementations, each offering unique features and optimizations tailored to specific use cases and performance requirements.

Implementations:

- **IBM System R** [3]
  - **IBM System R** used a primitive form of code generation and query compilation in the 1970s.
  - It compiled SQL statements into assembly code by selecting code templates for each operator.
  - The technique was abandoned when IBM built **SQL/DS** and **DB2** in the 1980s due to the high cost of external function calls, poor portability, and software engineer complications.
- **Vectorwise** [12]
  - **Vectorwise** pre-compiles thousands of ”primitives” that perform basic operations on typed data. Using simple kernels for each primitive means that they are easier to vectorize.
  - The DBMS then executes a query plan that invokes these primitives at runtime. Function calls are amortized over multiple tuples. The output of a primitive is the offsets of tuples.
- **Amazon Redshift** [2]
  - **Amazon Redshift** converts query fragments into templated C++ code. It's a push-based execution with vectorization.
  - DBMS checks whether there already exists a compiled version of each templated fragment in the customer's local cache.

  - If the fragment does not exist in the local cache, then it checks a global cache for the entire fleet of Redshift customers.
- **Oracle**
  - **Oracle** converts PL/SQL stored procedures into `Pro*C` code and then compiles it into native C/C++ code.
  - They also put Oracle-specific operations (e.g., compression, vectorization, and security) directly in the SPARC chips as co-processors.
- **Microsoft Hekaton** [4]
  - **Microsoft Hekaton** can compile both procedures and SQL. Non-Hekaton queries can access Hekaton tables through compiled inter-operators.
  - It generates C code from an imperative syntax tree, compiles it into DDL, and links at runtime.
  - It employs safety measures to prevent somebody from injecting malicious code in a query.
- **SQLite**
  - **SQLite** converts a query plan into opcodes and then executes them in a custom VM (bytecode engine). Opcode specifications can change across versions. This is also known as ”Virtual DataBase Engine” (VDBE)
  - **SQLite**'s VM ensures that queries execute the same in any possible environment.
- **TUM Umbra** [6]
  - **Umbra**'s ”FlyingStart” adaptive execution framework generates custom IR that maps to x86 assembly in a single pass. The developers manually perform dead code elimination. The DBMS is basically a compiler. They also wrote their own debugger!
- **Java Databases**
  - There are several JVM-based DBMSs that contain custom code that emits JVM bytecode directly, including Spark, Neo4j, Splice Machine, Presto / Trino, Derby, Tajo. This is functionally the same as generating LLVM IR.
- **Apache Spark** [1]
  - **Apache Spark** was introduced in the new Tungsten engine in 2015. The system converts a query's WHERE clause expression trees into Scala ASTs and then compiles these ASTs to generate JVM bytecode, which is then executed natively.
  - Databricks abandoned this approach with their new Photon engine in the late 2010s.
- **QuestDB**
  - **QuestDB** is a Java-based time-series columnar DBMS. The Java front-end converts WHERE clause predicates into IR and then uses a C++ backend to compile the IR into vectorized machine code using asmjit.
- **SingleStore**
  - Before 2016, **SingleStore** performed the same C/C++ code generation as HIQUE [9] and then invoked `gcc`. It also converts all queries into a parameterized form and caches the compiled query plan.
  - Since 2016, a query plan is converted into an imperative plan expressed in a high-level imperative DSL, MemSQL Programming Language (MLP), which gets converted into custom opcodes called MemSQL Bit Code (MBC). Lastly, the DBMS compiles the opcodes into LLVM IR and then to native code.
- **PostgreSQL** [8]
  - **PostgreSQL** added support in 2018 (v11) for JIT compilation of predicates and tuple deserialization with LLVM. It relies on optimizer estimates to determine when to compile expressions.
  - It automatically compiles Postgres' backend C code into LLVM C++ code to remove iterator calls.
- **VitesseDB**

  - **VitesseDB** is a query accelerator for **Postgres/Greenplum** that uses LLVM + intra-query parallelism with a push-based model. It also supports JIT predicates and makes indirect calls direct or inlined.
  - It does not support all of Postgres' types and functionalities. All DML operations are still interpreted.
- **CMU NOISEPAGE (2019)**
  - **CMU NOISEPAGE** uses SingleStore-style conversion of query plans into a database-oriented DSL, which is then compiled into opcodes.
  - It uses HyPer-style interpretation of opcodes while compilation occurs in the background with LLVM.

6 Vectorization vs. Compilation

Test-bed system [5] to analyze the trade-offs between vectorized execution and query compilation. High-level algorithms are implemented in both systems, but the implementation details vary based on the architecture of each system.

- **Approach #1: Tectorwise** It breaks operations into pre-compiled primitives. The output of primitives must be materialized at each step.
- **Approach #2: Typer** A push-based processing model with JIT compilation, where it processes a single tuple up the entire pipeline without materializing the intermediate results.

Both models are efficient and achieve roughly the same performance. (100x faster than row-oriented DBMSs!) Data-centric is better for ”calculation-heavy” queries with few cache misses. Vectorization is slightly better at hiding cache miss latencies.

7 Conclusions

Query compilation makes a difference but is nontrivial to implement. The 2016 version of SingleStore is the best query compilation implementation out there in terms of performance and engineering. Umbra FlyingStart is ridiculously good, but that's because the Germans are ridiculously good. Newer systems choose to implement Vectorwise-style vectorization instead of compilation.

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] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry. Amazon redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2205–2217, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526045. URL `https://doi.org/10.1145/3514221.3526045`.

[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] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13): 2209–2222, sep 2018. ISSN 2150-8097. doi: 10.14778/3275366.3284966. URL `https://doi.org/10.14778/3275366.3284966`.

[6] T. Kersten, V. Leis, and T. Neumann. Tidy tuples and flying start: fast compilation and fast execution of relational queries in umbra. *The VLDB Journal*, 30(5):883–905, jun 2021. ISSN 1066-8888. doi: 10.1007/s00778-020-00643-4. URL `https://doi.org/10.1007/s00778-020-00643-4`.

[7] 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`.

[8] 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`.

[9] 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`.

[10] 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`.

[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] 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, page 1231–1242, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465292. URL `https://doi.org/10.1145/2463676.2465292`.