Review Report-3

By Harsha Salim (USC ID: [redacted])

Paper: A. Kohn, et al., Adaptive Execution of Compiled Queries, in ICDE 2018

Summary

This paper discusses the challenges and solutions associated with compiling queries to machine code in database systems. The focus is on reducing the latency caused by query compilation, especially for interactive applications and large, complex queries. The authors propose an adaptive execution framework that dynamically switches between interpretation and compilation to achieve low latency for small queries and high throughput for long-running queries.

Compiling queries has been a popular alternative to interpreting queries during run time. By generating code for a query, it avoids the interpretation overhead of traditional execution engines. Hence it is very efficient in terms of performance, but it is not suitable for all types of queries. This is due to the overhead brought by the compilation process itself. For queries that touch a small number of tuples, or those that access small tables, compilation accounts for a significant portion of time taken by the query. There are also some extremely large queries used by business intelligence tools that cannot be compiled with standard compilers. For such workloads, a traditional interpretation-based engine was found to provide a better user experience in terms of execution time. Thus, depending on the query, one would prefer to have a compilation-based engines for some cases, and an interpretation based one for other cases. Implementing two query engines in the same system, however, would involve disproportionate efforts and may cause subtle bugs due to minor semantic differences. Hence the paper justifies a need for an adaptive execution framework, that is principally based on a single compilation-based query engine, while also integrating interpretation techniques that reduce the query latency.

The core ideas of this paper include a fast bytecode interpreter specialized for database queries, a method of accurately tracking query progress and, a way to dynamically switch between interpretation and compilation. Before getting into its design, the paper compares query execution and compilation to identify the scenarios where which operation would perform better. It ran related tests on TPC-H query 1 and concluded that for long running queries, compilation to machine code with a maximum optimization level is often preferable, while for quick queries, an interpreter would be best. The paper then delves into its proposed adaptive execution mechanism, which consists of three execution modes – bytecode interpretation for quick queries, unoptimized machine code for medium sized queries, and optimized machine code for long running queries. Instead of predetermining which mode to execute, which may rely on inaccurate cost estimates, which would result in an incorrect decision, the approach is dynamic, and we choose the mode based on

monitoring the execution progress. This fine-grained manner ensures that optimized compilation is done only for very expensive parts of the query.
The query progress is tracked, and its data dependent parts are performed in the worker functions, each of which compute one pipeline of the query and process the intermediate data and the range of values. Within a worker function, there is a morsel, which represents the smallest unit of work that can be processed by the query engine and are the mechanism to dynamically switch between different execution modes. For every morsel, the fastest available representation would be chosen as the next level of processing. There are three different options available for every pipeline, which corresponds to the three execution modes mentioned earlier. For bytecode interpretation, the response time is based on the current processing speed. The local tuple processing rate is monitored, and this is extrapolated to determine the total pipeline duration using the number of remaining tuples in the pipeline, and the number of active worker threads. While considering compilation, the remaining time in the pipeline depends on the expected compilation time, the execution time of the compiled code, and the possibility of using the idle threads for execution. The latter would require the computation of tuples that can be processed on the remaining threads during compilation and extrapolating the time needed for remainder. The compilation time and the corresponding execution time depends on the generated query plan. However these are rough extrapolations, which are sufficient. After every morsel, the thread computes the average processing speed of all threads and compares the remaining processing time of the execution modes. If the transition to a new execution mode appears beneficial, the thread compiles the worker function and resets all processing rates. This allows one to eventually transition to the fastest execution mode for every pipeline.

LLVM's interpreter is extremely slow, due the cache unfriendliness of its pointer-based-in-memory representation, and the costly runtime dispatch of its instructions. To make this more viable, the native intermediate representation of LLVM was translated into an optimized bytecode format for a virtual machine. The resultant challenges are to ensure cheap bytecode processing, efficient translation to the bytecode, and the VM behaving identical to the native machine code. The virtual machine used here is a register machine, where register files hold all the values computed during function allocation. A switch statement converts each LLVM instruction to the corresponding bytecode operations. In some cases, multiple LLVM instructions would be collapsed to one VM instruction to handle frequently occurring instruction sequences. The LLVM programs are in Single Static Assignment form wherein a value is produced exactly once, and it never changes during the lifetime of the program. This makes translation easy, as we just have to allocate registers to the live values in a block. Hence the translation of LLVM IR code into VM code starts by computing the liveness information of the values in a block. Once registers are allocated for the live values in each block, the instructions in that block are translated into VM opcodes one by one, excluding the ones that are subsumed by previous instructions. At the end of the block, the values are copied into ϕ nodes of successor blocks if needed, and registers are released for all values where the lifetime has ended. Liveness computation is based on the following 2 concepts: liveness is computed as a live-range with a start and an end block, and the live-range of each value is kept as tight as possible. The interoperability

between bytecode and machine code, raises a problem. While a function pointer suffices to run machine code, the bytecode has to be interpreted using the virtual machine. So instead of a direct function call, an additional dispatch code needs to be called with the function's bytecode as an additional argument. The paper proposes to always have an extra pointer argument to the function even though it is redundant in the machine code case, as it allows for transparent switching between the two modes. The paper also suggests some optimizations that speed up processing of commonly occurring operations.

The experimental results demonstrate that the adaptive execution framework consistently outperforms static execution modes across various scenarios, particularly for queries in the TPC-H benchmark. They also showed that adaptive execution more effectively utilized the threads in a multithreaded system. The paper even provides evidence to their justification for the adaptive execution model, by running tests to show that machine code compilation is very slow compared to planning and code generation, with optimized compilation being even slower, while bytecode generation is very fast. Similarly, they also compare the execution times of the bytecode interpreter and the compiled machine code, to show that the bytecode interpreter is slower than the compiled machine code. While testing with very large queries, the paper concluded that fast bytecode translation is indispensable in such scenarios and compilation takes a very long time in the same situation. A system with adaptive execution effectively adapts to data sizes ranging from 10MB to 30GB, showcasing its capability to balance latency and throughput dynamically. The findings confirm that both interpretation and compilation are crucial for optimizing query execution times in database systems.

Paper Strengths

The paper clearly makes a case for why we need adaptive execution of compiled queries. It starts by explaining the pros and cons of compilation and interpretation and showcases through tests that both perform well under different scenarios. Therefore, it emphasizes the reasoning for a combination of the two to be used to satisfy real world queries. It addresses the critical issue of query compilation latency in database systems, a problem that significantly affects the performance of interactive applications that require rapid responses. The research question is highly relevant, as it targets a common bottleneck in database management, particularly for complex queries. This proposed framework allows for a dynamic balance between low latency and high throughput, resulting in a very efficient system.

One of the core contributions of the paper is the adaptive execution framework that leverages real-time feedback to switch between interpretation and compilation modes. This approach is novel because it allows the system to optimize performance based on the complexity of the query, its progress and the size of the data being processed. The fast bytecode interpreter that translates LLVM Intermediate Representation (IR) into a compact bytecode format reduces overhead associated with traditional compilation methods. Furthermore, the dynamic execution mode switching mechanism enables the system to

make real-time decisions about execution strategies, which is a significant departure from static execution models commonly used in existing database systems. Regarding these components, the paper goes into great depth to with respect to the design and implementation along with examples to ensure that the readers are able to easily understand the concepts.

The paper appears to be quite well rounded when proposing a certain method over others. This has been observed in the paper during multiple instances, where they acknowledge the cons of introducing the proposed method, and not just focus on the benefits it brings. Furthermore, they evaluate the extent of overhead/cost incurred by that cost and decide if that is a viable tradeoff considering the benefits. For instance, the bytecode interpretation done by the virtual machine is enhanced by fixed length encoding for opcodes. While this increases the decoding speed, it also increases the memory footprint of the translated function relative to native machine code. This was brought up when this optimization was suggested to provide the reader a full picture. Following this, they also evaluated that the space occupied by this is still more compact than the pointer-heavy LLVM IR (Intermediate Representation).

The paper provides comprehensive analytical and experimental findings that demonstrate the effectiveness of the proposed framework. The results indicate that the adaptive execution framework consistently outperforms static execution modes across various scenarios, particularly in the TPC-H benchmark tests. The system's ability to adapt dynamically to data sizes ranging from 10MB to 30GB showcases its robustness in handling different query demands while balancing latency and throughput. The clarity of data analysis and presentation further strengthens the paper, making it accessible to readers while conveying complex ideas effectively. The paper also includes additional tests to emphasize the justification for adaptive execution. Instead of just stating observations about compilation and bytecode interpretation, the paper also includes experiments to prove this.

The findings of this paper hold significant implications for future research and practical applications in database management. By demonstrating the importance of both interpretation and compilation in optimizing query execution times, the authors pave the way for further exploration of adaptive execution strategies in various contexts. This research could inspire new methodologies for enhancing database performance, particularly in environments where rapid data processing is critical. The well-organized structure and comprehensive literature review within the paper also provide a solid foundation for future studies, making it a valuable resource for researchers and practitioners alike.

Paper Weaknesses

When evaluating the limitations of this study on adaptive execution frameworks for compiled queries, several key areas warrant attention. First, the methodology employed for

register allocation and liveness computation presents potential inefficiencies. Although the study introduces a new linear time register allocation algorithm, it acknowledges that this approach may extend the lifetime of variables unnecessarily within certain complex control flows. Such a trade-off, while aimed at maintaining linear runtime efficiency, could lead to suboptimal register usage and could impact the performance of very large queries that exhibit intricate control structures. This also showcases that the paper seems heavily focused on reducing the total time required by the queries, but this is at the cost of space. There could be scenarios where space may also need to be optimized.

The experiments focus primarily on TPC-H and TPC-DS benchmarks, which, while widely used, may not fully capture real-world workloads, especially those with irregular query patterns and mixed OLTP/OLAP characteristics. There is limited discussion on how adaptive execution performs on ad-hoc queries beyond analytical benchmarks. Machine-generated queries are mentioned but not explored in depth regarding their diversity and impact on adaptive execution.

The paper assumes that LLVM-based compilation is the dominant approach for query execution but does not compare its findings with alternative compilation frameworks or execution models, such as vectorized engines. It does not mention the alternatives during the explanation of the design and implementation, nor is it mentioned during the evaluation, which would have further helped the reader to make a more informed decision.

Lastly, the study's reliance on empirical evaluations conducted on specific hardware configurations raises questions about generalizability. The experiments were performed on a desktop system with an AMD Ryzen 7 CPU, and repeated on an Intel CPU, and may not reflect performance on other architectures or under different operational conditions. This limitation suggests that further testing across diverse environments would be beneficial to validate the findings and ensure robustness in various deployment scenarios

In summary, while the study presents innovative approaches to enhancing query execution through adaptive techniques, it is essential to consider these limitations to fully understand the implications of its findings.

Opinion

Despite the weaknesses mentioned, the paper does a stellar job of explanation the need for and the implementation of adaptive execution and its related components. The evaluations at the end also clearly showcase the benefit it brings to query processing performance. Hence, I accept this paper.