Parallel LLVM Execution Engine
15-712 Advanced Operating Systems and Distributed Systems

Zhixun Tan
zhixunt@andrew.cmu.edu

Hao Jin
haoj@andrew.cmu.edu

***Abstract*—We extended the LLVM query compilation feature in Peloton, an in-memory database system developed at CMU. In particular, we parallelized query execution for TableScan and HashJoin. We adopted the morsel-driven parallel execution model, where target tables are split into chunks and processed in separate threads. This parallel execution model is well suited for the push-based query compilation model, where operators in a pipeline can be fused into a single scanning loop.**

Keywords*—*Query Compilation, Parallel Execution, Database.

I. Introduction

Normally a database system works like an **interpreter** of SQL queries: The user provides a SQL query as a string, then the system parses it into an Abstract Syntax Tree (AST), performs annotations, analyses and transformations on the AST, and executes corresponding operations. The part of the system that is responsible for executing the AST is called the “execution engine". The execution engine can be thought of as a function that takes in an AST and outputs the result.

Traditional database systems use the pull-based Volcano [1] iterator model. Each operator implements a `GetNext()` method, which recursively calls the `GetNext()` method of its child(ren). The entire AST is therefore traversed once each time a tuple gets outputted.

An alternative idea is that, instead of interpreting the AST, we can compile and then execute it. In this way, the compiled code is specialized for that specific query, thus achieving better performance. There are mainly two approaches for query compilation: generating C/C++ code and invoking a C compiler, or generating LLVM IR and utilizing the LLVM JIT engine. Previous experiments [2] show that the first approach introduces too much compilation overhead. Therefore, the second approach is more widely adopted. HyPer [2], MemSQL [3], Cloudera Impala [4], and Microsoft Hekaton [5] are some examples of real-world database systems that have the query compilation feature.

Query compilation reduces interpretation overhead in the following ways:

- When performing expression evaluation, the interpreter must traverse the expression tree once for every tuple, which involves many virtual function calls. With query compilation, with the knowledge of all the types, the generated code directly uses the corresponding machine instructions.
- Query compilation eliminates the need to recursively traverse the operator tree. Operators in the pipeline are effectively “fused" together in a single piece of code.
- Query compilation eliminates tuple materialization between operators in a pipeline. By generating code for the entire query, each tuple only needs to be loaded and materialized once in each pipeline.

Peloton is a database system that is now being developed at Carnegie Mellon University. Its execution engine is currently in the process of transitioning from a interpreted one to a compiled one. A number of features are already supported, including basic table scan, hash join, aggregates and expression evaluation. For our project, we implemented parallel execution in this compiled engine. More specifically, we generated multithreaded code for processing queries.

Peloton uses full-query compilation. It fuses operators in a pipeline to a single loop. Only queries that must be executed in separate stages (such as HashJoin that must be executed by first building a hash table from the first table and then probing it from the second table) do we compile into multiple loops. This suggests that we can naturally adopt a data-parallel model. We split target tables into chunks, and submit tasks that are responsible for different chunks to a task queue. And a thread pool will be responsible for running the tasks. A key property that we maintain is that each task still runs the entire pipeline - we are not putting different operators, but different chunks of the table into different threads.

The rest of this report is as follows. We describe related work on both query compilation itself as well as parallel query execution in sections II and III. Then we use concrete examples to explain the two kinds of queries we parallelized - TableScan and HashJoin. In section IV we discuss how TableScan is compiled and how we parallelize it. In section V we discuss how HashJoin is compiled and how we paralleize it. Then we discuss performance evaluation in section VI. Finally we conclude this report in section VII.

II. Related Work - Query Compilation

Query compilation dates back to as early as the 1970s, when IBM System R [6] compiled SQL queries into assembly code by applying a corresponding code template for each operator. However, it is later abandoned for poor performance, lack of portability, and significant engineering complexity. Then, query compilation was not widely adopted for major database systems, mainly because in traditional disk-oriented databases, the performance overhead was disk I/O, and the benefit of eliminating interpretation was little.

Recent in-memory database systems have adopted query compilation in different ways.