Predicting Query Execution time for JIT Compiled Database Engines

Kostas Chasialis [1], Srinivas Karthik [1], Bikash Chandra[1#], Anastasia Ailamaki[1,2]
1: EPFL 2: Raw Labs

Predicting query execution time in a database is essential in a wide variety of scenarios such as query scheduling, resource allocation, query progress monitoring, and admission control of queries. Modern database systems use in-memory data processing and incorporate novel technologies such as code generation. These systems present new challenges and opportunities for accurate prediction of execution times. For instance, a majority of the total execution time for analytical queries is spent on memory access and branch misprediction [2].

Prior works in query performance prediction have focused on using hefty machine learning - requiring large training times before the models become usable - or lightweight analytical models which are highly inaccurate. In this paper, we propose a novel light- weight analytical cost model, `JIT-Prediction`, using active learning for predicting query execution times for JIT code generation based systems, as described next.

JIT-PREDICTION. We build on the analytical model introduced in [1] as to support ad-hoc queries. The key idea in `JIT-Prediction` is to learn many of the underlying *hidden* architecture parameters through *active learning* – by executing a carefully designed set of calibration queries based on query data access patterns. Through these queries, we introduce a *fudge factor* to the erroneous parameters in the model, and infer them mathematically. These fudge factors essentially play a role of corrective measure for the errors induced by the model. This phase is designed to take a small fraction of the total execution time, as opposed to ML techniques which run for several hours.

Let us see `JIT-Prediction` for a simple scan query which uses *Sequential Traversal* (S_{trav}). As the name suggests, the underlying access pattern on a memory R, sequentially sweeps over R, accessing the contiguous blocks in the memory. For a 3-level cache, the memory access time, T_{mem}, is given by

$$T_{mem} = M_1^s * l_2^s + M_2^s * l_3^s + M_3^s * l_4^s \quad (1)$$

Here, M_1^s, M_2^s and M_3^s are the estimated cache misses from the Manegold Model [1]; l_2^s, l_3^s and l_4^s are the sequential access latencies for L2 cache, L3 cache and DRAM, respectively.

In order to infer the fudge factors, we use the projection-based calibration queries which emit only S_{trav} of the form:
select sum(attr[s]) from table;.

We introduce the fudge factors, F_i for each cache level i, as a corrective measure for cache misses. Then the new equation is

$$T^a_{mem} = \mathsf{F}_1 * M_1^s * l_2^s + \mathsf{F}_2 * M_2^s * l_3^s + \mathsf{F}_3 * M_3^s * l_4^s \quad (2)$$

# Currently at Meta Platforms, Inc.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2023. 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

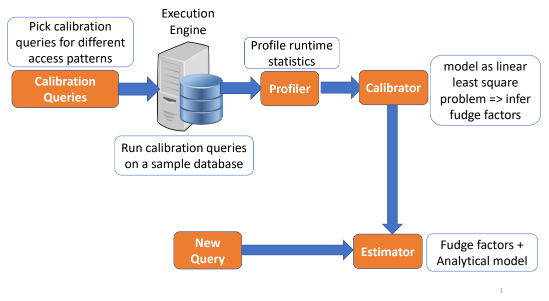
Pick calibration queries for different access patterns
Calibration Queries
Execution Engine
Run calibration queries on a sample database
Profile runtime statistics
Profiler
Calibrator
model as linear least square problem => infer fudge factors
New Query
Estimator
Fudge factors + Analytical model

Figure 1: `JIT-Prediction`**'s Workflow**

We obtain the actual memory access time, T^a_{mem}, using the Intel VTune profiler. Since we need a basic profiler, other execution profiling tools can also be used. The 3 unknowns in Equation 2 can be solved by running 3 projection queries. However, since the measured results are somewhat noisy, we model the problem as a *linear least-squares problem with bounds on the variables*, and solve it using *Trust Region Reflective algorithm.*

This is the first step in our offline calibration phase, other operators which contains more complex access patterns are calibrated progressively. For example, hashing and sorting induce random access patterns. Hence, Equation 1 is extended to include random misses and latency – `JIT-Prediction` also infers fudge factors for these additional terms. With filters, cost from branch misprediction plays an important role, which is handled with `JIT-Prediction` similarly. We omit the details here in the interest of space.

Results. We evaluate our proposed technique, `JIT-Prediction`, on SPJ queries from the SSBM Benchmark, and compare against the actual execution time and the state-of-the-art approach, i.e. Manegold Model [1]. For our technique, all the access patterns work in combination, and the inferred fudge factors from the calibration queries (capturing individual access patterns) are applied to these queries. With the calibration, the overall average Q-error reduces by 93% in comparison to the Manegold Model, and the average Q-error for `JIT-Prediction` is 2.26 in comparison to the ideal. We also performed experiments to show that the fudge factors are nearly scale-independent. Thus, one can apply the calibration phase on a smaller sample of the database.

Future Work. Extending the framework to include more workloads and execution environments such as scale-up, scale-out, or even with accelerators such as GPUs is an area of future work.

REFERENCES

[1] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Generic Database Cost Models for Hierarchical Memory Systems. In *PVLDB*. 191–202.
[2] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *PVLDB* 13, 6 (2020), 840–853.