One Loop Does Not Fit All

Styliani Pantela
Harvard University
stylianipantela@college.harvard.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

ABSTRACT

Just-In-Time (JIT) compilation increasingly becomes a key technology for modern database systems. It allows the creation of code on-the-fly to perfectly match an active query. In the past, it has been argued that a query should be compiled to a single loop that performs all query actions, for example, all selects over all relevant columns. On the other hand, vectorization – a common feature in modern data systems – allows for better results by evaluating the query predicates sequentially in different tight for-loops.

In this paper, we study JIT compilation for modern in-memory column-stores in detail and we show that, contrary to the common belief that vectorization outweighs the benefits of having one loop, there are cases in which creating a single loop is actually the optimal solution. In fact, deciding between multiple or a single loop is not a static decision; instead, it depends on (per column) query selectivity. We perform our experiments on a modern column-store prototype that supports vectorization and we show that, depending on selectivity, a different code layout is optimal. When a select operator is implemented with a no-branch design, for low selectivity creating multiple loops performs better than a single loop. A single tight loop performs better otherwise.

1. INTRODUCTION

Just-In-Time Code Generation. JIT code generation for query execution is a method employed by database management systems in order to avoid the interpretation overhead that comes with declarative languages like SQL [6]. Upon receiving the query, the engine makes decisions on optimization and compiles SQL into a low level representation that will then be linked, loaded and executed. This method avoids pitfalls that could be incurred by the separation of operators, a key insight exploited in HyPer [3]. HyPer uses JIT code generation for an in-memory column store system by avoiding intermediate materialization of results until a pipeline breaker appears. The generated code breaks operator boundaries combining together as many operators as possible in a single loop. It keeps intermediate results in registers and generates code in LLVM IR and suggests that *one loop fits all*.

Furthermore, Sompolski et al. [5] extensively study the ideas of JIT compilation and vectorized query processing in column stores and prove that it is beneficial to introduce JIT query compilation to a vectorized system and that there are still benefits of vectorization to a tuple-at-a-time compiled system. The premise of JIT code generation in a compact loop is that it avoids the creation of intermediate results [5]. Sompolski et al. argue that if we reduce the generation of intermediate results by using a select_fetch operator instead of using a fetch followed by a separate select, the approach of using one loop is worse that simply evaluating one predicate after another sequentially, in a vectorized way.

Contributions. In this paper we experiment with modern in-memory column store systems and look closely at the case of conjunctive selects. We show that in a multi-threaded setting, it is beneficial to switch to the one loop technique most of the time. However, in cases of low selectivity and branchless execution it is beneficial to opt for having multiple loops.

2. ONE LOOP DOES NOT FIT ALL

In this section we present our experimental analysis and we demonstrate that a single JIT solution is not optimal across all workload scenarios.

Experimental Setup. We use an in-memory column store prototype. Every column is assumed to fit in main memory and the tuples are 8 bytes long for a total of one billion tuples per column. We evaluate our system on a 4-way Intel Xeon E7-4820 configuration with 64 hardware threads and 1 TB of main memory and use GCC 4.7.2 (Debian 4.7.2-5).

Test Queries. We focus our experiments on conjunctive select queries of the form of Query 1. Such queries stress the decision between a single and multiple loops, as they include several predicates.

Query 1 Select Query

```
SELECT R.d FROM R
WHERE R.a < v1 AND R.b < v2
AND R.c < v3 AND R.d < v4
```

Execution Strategies. The first strategy of execution breaks the evaluation of the query into four tight loops; one for each predicate. In this case vectorized execution can be exploited. Each loop operates on a vector of tuples that fits into the L1 cache and performs the select over the respec-

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2764944.

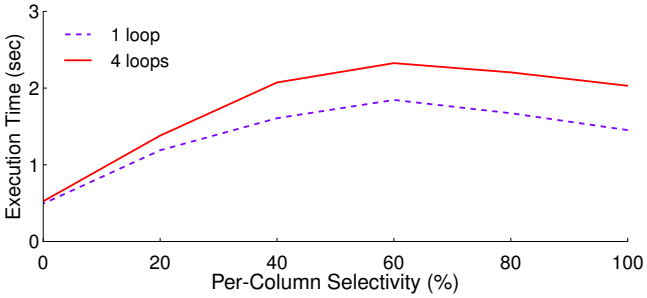

Figure 1: With branches

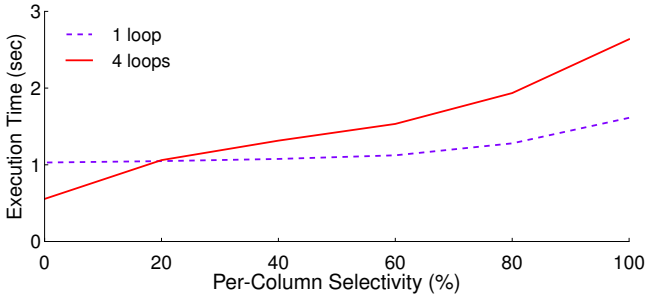

Figure 2: Branchless

tive column using column store operations [1]. After the first select, we can minimize intermediate results by using a fetch_select operator to perform a column projection and selection in one step, following prior art [2] (Algorithm 1).

Algorithm 1 *4 Loops Strategy* for Query 1

```
1. inter = select(R.a, v1)
2. inter = select_fetch(R.b, v2, inter)
3. inter = select_fetch(R.c, v3, inter)
4. dest  = select_fetch(R.d, v4, inter)
```

The second execution strategy uses one tight loop for the evaluation of all of predicates (Algorithm 2).

Algorithm 2 *1 Loop Strategy* for Query 1

```
1. dest = select(R.a, v1, R.b, v2,
                      R.c, v3, R.d, v4)
```

Optimizations. Multi-threaded execution helps to improve performance; we fully utilize the eight available cores by using eight threads. Following prior work [4] we evaluate the two execution strategies with two different implementations: with and without branching. The order of evaluation of the predicates is inconsequential because the data is randomly generated using a uniform distribution, and the per-column selectivity is the same across all predicates. Finally, we use loop unrolling to optimize the branchless version of vectorized processing.

Results. Figure 1 shows the performance of the two execution strategies for different column selectivity in an implementation that uses branching. The performance is affected by branch mispredictions as selectivity varies. The *1 Loop Strategy* (Algorithm 1) outperforms the *4 Loops Strategy* (Algorithm 2) for any selectivity and the gap is increased in high selectivity. Furthermore, Figure 2 compares the two execution strategies in a branchless context. The *1 Loop Strategy* is outperforming the *4 Loops Strategy* for higher selectivity but is slower for lower selectivity.

Although it is argued that vectorization comes with a lot of benefits [5], the functional overhead we pay along with the cost of materializing intermediate results is significant in both the branchless and the with cases. The intuition behind our observation is that in the *1 Loop Strategy*, all columns of the predicates will be scanned fully regardless of the selectivity. However, for low selectivity, most of the tuples across the columns are not going to qualify. This results in reading data that is not going to be used. On the other hand, using separate loops for each predicate results in scanning fewer cache lines when the first filter selects very few tuples.

Comparing Figure 1 and 2 we observe that optimizing for performance requires the branched execution (especially for low selectivity), where we should always opt for the *1 Loop Strategy*. On the other hand, the branchless version yields more predictable performance. For low selectivity the *4 Loops Strategy* outperforms the *1 Loop Strategy*, and vice versa for high selectivity.

Finally, we observe that the difference between the two execution strategies in Figure 1 and Figure 2 is increasing and appears to be dominated by the extra cost of the creation of intermediate results for the vectorized execution.

3. SUMMARY & FUTURE WORK

We study code generation for in-memory column store systems, covering the case of conjunctive selects. We show that there is no single solution that works optimally across all workloads; selectivity is the crucial factor. Our conclusion is that a robust system using branchless execution should choose to alternate between the two methods based on predicted selectivity.

Future steps include a more detailed breakdown of hardware metrics like CPU cycles and branch mispredictions for varying selectivity in order to further support our findings on which strategy performs better. Another open question is whether employing SIMD instructions when comparing the two strategies will yield different results. In order to establish a general heuristic, we plan to extend our analysis by evaluating disjunctive queries, and varying data distribution and correlations across columns.

Acknowledgments. This project is supported by the National Science Foundation under Grant No. IIS-1452595. The authors would like to thank Lukas Maas and Manos Athanassoulis, members of Harvard DASlab, for their help.

4. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5:197–280, 2013.

[2] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007.

[3] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[4] K. A. Ross. Selection Conditions in Main Memory. *ACM Transactions on Database Systems*, 29(1):132–161, 2004.

[5] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. Compilation in Query Execution. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 33–40, 2011.

[6] S. D. Viglas. Just-in-time Compilation for SQL Query Processing. *Proceedings of the VLDB Endowment*, 6(11):1190–1191, 2013.