Vectorizing an In Situ Query Engine

Panagiotis Sioulas
University of Athens
panossioulas@gmail.com

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
anastasia.ailamaki@epfl.ch

ABSTRACT

Database systems serve a wide range of use cases efficiently, but require data to be loaded and adapted to the system's execution engine. This pre-processing step is a bottleneck to the analysis of the increasingly large and heterogeneous datasets. Therefore, numerous research efforts advocate for querying each dataset *in situ,*i.e., without pre-loading it in a DBMS. On the other hand, performing analysis over *raw data* entails numerous overheads because of the potentially inefficient data representations.

In this paper, we investigate the effect of vector processing on raw data querying. We enhance the operators of a query engine to use SIMD operations. Specifically, we examine the effect of SIMD on two different cases: the scan operators that perform the CPU-intensive task of input parsing, and the part of the query pipeline that performs a selection and computes an aggregate. We show that a vectorized approach has a lot of potential to improve performance, which nevertheless comes with trade-offs.

1. INTRODUCTION

As the size of data and its variety continue to grow at rapid rates, loading entire datasets in a DBMS prior to querying them constitutes a bottleneck, incurred before any queries can be launched over the datasets. In addition, many practitioners avoid the use of DBMS altogether due to vendor lock-in concerns. They avoid storing their data in proprietary, DBMS-specific file formats because they still want to launch external scripts or other programs over their data. Consequently, numerous systems advocate querying data in their raw representation, without loading them in a DBMS a priori [1,2,3,4,5].

On the other hand, DBMS load data in a compact, well-engineered representation to minimize access costs. Querying raw data introduces various overheads. For verbose, textual file formats, every query has to re-parse the input data, identify tokens per "tuple ", convert raw data fields, etc. To minimize the costs, systems querying raw data use various techniques, such as specialized index structures [1,2,4] and parallelism [3,5]. The former techniques target scenarios where data accesses are meant to be judicious, whereas the latter attempt to load the raw data with a negligible loading cost.

This work focuses on how judicious data accesses can be further sped up through the use of vectorization [5]. We therefore use a pipelined query engine as the starting point, and extend it to be able to access raw CSV data with the use of *positional maps*[1]. Positional maps are auxiliary structures that capture the position of raw data fields in a CSV file. For example, for every row in a CSV file, the positional map for it captures the positions of the 1st, 6th, 11th, … fields. The system can thus use the known positions to navigate in the raw file with reduced cost, avoiding parsing from scratch. Then, we extend the query engine with SIMD processing primitives to further reduce processing costs, which allow performing the same operation for a vector of data at a time, thus taking advantage of data level parallelism.

Contributions. In this paper, we examine SIMD optimizations on a query engine for raw data. We develop a SIMD-powered scan operator for CSV data, and an aggregation operator that also performs data filtering. Our results show that SIMD-aware operators outperform their scalar counterparts in the majority of cases. There are cases, however, where combining scalar and vectorized code is more beneficial.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).
SIGMOD'16, June 26 - July 01, 2016, San Francisco, CA, USA
ACM 978-1-4503-3531-7/16/06.
http://dx.doi.org/10.1145/2882903.2914829

2. OPTIMIZED OPERATORS

Scan. The query engine we use converts judiciously only the data fields necessary to answer the current query. Therefore, a typical invocation of a scan operator for CSV files involves a combination of field skips and conversions of target fields per CSV record. A skip searches for a delimiter, either forwards or backwards in the file, using a character-to-character comparison; the delimiter denotes the end of a field. The single-character comparisons can be vectorized by loading N bytes at a time in SIMD registers and comparing them in vector-sized chunks [5]. The result of the SIMD comparison is a mask vector that can be converted to a single integer bitmask, with set bit positions corresponding to delimiter offsets. The number of trailing zeroes in this bitmask equals the offset of the delimiter. The search for the next field continues from the last delimiter found.

For relatively short fields, consecutive loads of the SIMD registers and comparisons overlap. The result is performance degradation, because we have to fill the SIMD registers repeatedly for short fields. As bitmasks hold the position of all delimiters in a chunk, this inefficiency can be avoided by reusing the bitmask's data rather than discarding it. A persistent bitmask can be used as previously, shifted appropriately for each skip to reflect the current position until it is depleted. Then, the next bitmask is computed. For instance, a "0001001001000000" bitmask produced by a 16 byte scan can be exploited for 3 skips, each consuming up to the rightmost set bit by shifting 7, 3 and 3 bits to the right respectively, before scanning forward again. This method implies a one-way traversal compared to the scalar version.

SIMD-aware scans are mostly useful when fully converting the data input. When, however, they are combined with a positional map for judicious data accesses, the persistent bitmask is always relative to the current position in the file and is made irrelevant by jumps. For this approach to operate correctly, the bitmask info has to be reset after using the index. By accessing data only to retrieve specific values, the code path becomes more complex. Thus, the benefit from vectorization is smaller.

Aggregations. Analytical queries typically include a series of filtering predicates and the calculation of some aggregate expressions. We therefore examine an operator that combines a selection and an aggregation step. For each record, the necessary columns are loaded, the condition is computed and, if true, the code branches to the aggregation. We have added a SIMD-aware

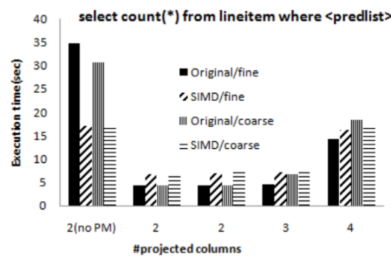

Figure 1: Scan w/ coarse/fine grained PM

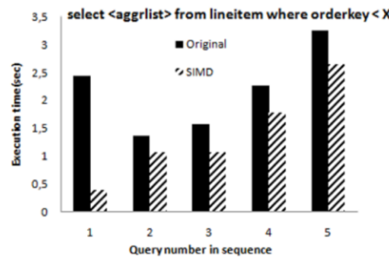

Figure 2: Aggregation pipelines

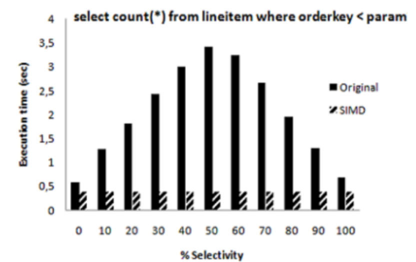

Figure 3: Misprediction sensitivity

version of the aggregation operator, which works with a vector of data at a time.

SIMD, however, is incompatible with aggregation by branching. The vectorized alternative is predication. The selection predicate results in a mask vector, applied to the entire input. In the end, the elements of the partial results are aggregated together to produce the result. As the overall operator supports multiple aggregations at once, this process takes place for each one of them.

3. EXPERIMENTAL EVALUATION

Experimental Setup. We evaluate the impact of our optimizations to the raw data querying engine using a server with 2 Intel Xeon E5-2650L v3 processors running at 1.8 GHz. These processors support 256-bit wide AVX2 SIMD instructions .The system has 384GB of DRAM and 2TB SATA3 HDD. We use the *LineItem* table from the TPC-H benchmark as the schema for the input datasets.

Scan. In this experiment, we evaluate the vectorized version of the scan operator. The input file uses the CSV file format and contains 60 million rows that require 7.2GB on disk. The workload consists of 5 queries and we clear the caches before running the first query. The first 3 queries in the sequence project 2 columns, while Q4 and Q5 project 3 and 4 columns respectively. We repeat the experiment with a positional map that contains one entry per 3 columns, and a coarser-grained version that uses one entry per 5 columns.

Figure 1 plots the execution time per query using either of the fine-and coarse-grained settings. In both cases, the positional map is populated during the execution of the first query and reused subsequently. The first query benefits greatly from the use of SIMD instructions, which decrease the execution time by 1.8-2x depending on the type of the positional map. However, the next two queries that project the same columns are 60% slower when using SIMD. These queries represent the ideal case for the original version of the scan operator as they can fully reuse the positional map for direct access to the raw data. On the other hand, the SIMD version encounters short sequences that create delays in the pipeline. When projecting additional columns, the benefits of SIMD depend on the position of the attributes relative to the state of the positional map. This trade-off can be observed in the query that projects 4 columns. Overall, utilizing SIMD efficiently requires tuning all parameters of the scan operator.

Aggregations. In this experiment, we evaluate the vectorized version of the aggregation pipeline. The input files use a binary columnar representation of the table with 600 million rows. The workload consists of 5 queries and we clear the caches before running the first query. All queries apply a filtering predicate with 30% selectivity on the *orderkey* attribute, and also vary the type of the aggregation operator and the columns it accesses. Q1 uses a simple "COUNT(*)" aggregation while Q2 uses "MAX(quantity)". Q3 combines the two aggregation operators, while Q4 and Q5 each add additional MAX operator on a different column to the preceding query in the sequence.

Figure 2 plots the execution time of the original and vectorized version of the system for the sequence of queries. In all cases, the SIMD version reduces execution time by a factor of 1.2-6x. The biggest improvement is observable in the Q1 COUNT query. This effect is due to the very high branch misprediction rate of 0.447 for the original version, whereas the SIMD version does not perform branches. High branch misprediction causes the execution time of Q1 to be higher than Q3, even though Q3 is essentially an extended version of Q1 that calculates more aggregates. We further examine Q1 in Figure 3, where we vary selectivity: For selectivities closer to 50%,we observe even higher performance improvement, which corroborates the finding of a related study [7]. When the pipelines include multiple operators, this effect is masked by other instructions required to compute the additional aggregations. Overall, the use of SIMD improves robustness by removing sensitivity to selectivity.

4. CONCLUSIONS

In this paper, we quantify the opportunities for applying vectorization techniques to raw data querying. We study two components of the system, the scan operator and the aggregation pipeline, and examine the trade-offs involved in using SIMD-powered vectorized execution. We show that the performance of the vectorized scan operator depends on both the positional map settings and the attribute layout, while the vectorized aggregations are more robust. In the future, we will apply vectorization techniques to other operators and fine-tune them to amortize the trade-offs.

5. REFERENCES

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos and A. Ailamaki. NoDB:Efficient Query Execution on Raw Data Files. In SIGMOD, 2012.

[2] S. Blanas, K. Wu, S. Byna, B. Dong and A. Shoshani. Parallel data analysis directly on scientific file formats. In SIGMOD, 2014.

[3] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In SIGMOD, 2014.

[4] M. Karpathiotakis, M. Branco, I. Alagiannis and A. Ailamaki. Adaptive Query Processing on RAW Data. In PVLDB, 7(12): 1119-1130, 2014.

[5] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper and T. Neumann. Instant loading for main memory databases. In PVLDB, 6(14): 1702-1713, 2013.

[6] O. Polychroniou, A. Raghavan and K.A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In SIGMOD, 2015.

[7] J. Sompolski, M. Zukowski and P. A. Boncz. Vectorization vs compilation in query execution. In DaMoN, 2011.