Run-Time Data Analysis in Dynamic Runtimes*

Lukas Makor
Johannes Kepler University
Linz, Austria
lukas.makor@jku.at

Abstract

Databases are typically faster in processing huge amounts of data than applications with hand-coded data access. Even though modern dynamic runtimes optimize applications intensively, they cannot perform certain optimizations that are traditionally used by database systems as they lack the required information. Thus, we propose to extend the capabilities of dynamic runtimes to allow them to collect fine-grained information of the processed data at run time and use it to perform database-like optimizations. By doing so, we want to enable dynamic runtimes to significantly boost the performance of data-processing workloads. Ideally, applications should be as fast as databases in data-processing workloads by detecting the data schema at run time. To show the feasibility of our approach, we are implementing it in a polyglot dynamic runtime.

Keywords: Data Analysis, Dynamic Runtime, Performance, Optimization

1 Motivation

Nowadays, data has become an important resource, but data is only valuable if information can be extracted from it. Often databases are used to handle the huge amounts of data that are generated, collected and processed. However, not only the amount of data, but also memory capacity and the processing power of computers has grown dramatically in recent years [1, 2]. Hence, more and more data is processed directly in memory instead of in a database. For in-memory data processing, most developers want to use the comfortable means available in programming languages and leave performing optimizations to the runtime. Consequently, optimizing such data processing patterns is becoming more important for dynamic runtimes.

However, a runtime typically is not able to perform database-like optimizations as it lacks the required information. Therefore, we propose an approach that entails extending dynamic runtimes to enable such optimizations by collecting information about the processed data at run time. One optimization is to adapt the way the data is stored based on the most relevant usages. This optimization can be performed by tracking the access patterns of individual data structure instances at run time and is discussed in more detail in Section 3.

Another optimization is condition reordering, which becomes possible when tracking and analyzing information about the processed data at run time. We can reorder the selection conditions inside a loop that, e.g., filters the data based on multiple properties based on the selectivities of the conditions to schedule the most selective ones first. Thus the evaluation of the remaining conditions is skipped for many elements, and the application's performance is improved.

The goal of our approach is to speed up the processing of large amounts of (semi-structured) data in a dynamic runtime without requiring users to perform additional actions or to adapt the way they write code. To accomplish this goal, the work on this project is split into two parts. This work deals with data collection, analysis, and high-level optimizations in a dynamic runtime, while the other part, which is developed concurrently, deals with optimizations in a dynamic compiler based on the collected information.

2 Problem

Processing huge amounts of data without putting attention to efficient data storage and access is not feasible, even with the processing power of today's computers. Hence, databases employ various optimization techniques such as query plan optimization [3, 4, 5], storage optimization [6, 7, 2], and automatic indexing [8, 9] to speed up data processing. Unfortunately, a runtime typically is not able to perform such database-like optimizations, because it lacks the information required to perform them. For example, it typically lacks information regarding 1) the structure of the objects stored in data structures, 2) the value distribution of the properties of the stored elements and 3) the run-time access patterns.

To the best of our knowledge, there are no approaches that collect run-time information about the actually processed data in database-like workloads and then perform optimizations based on this information. Of course, there are approaches that utilize run-time information to optimize program performance. Suganuma et al. [10] use run-time information to perform optimizations in a JIT compiler. Additionally, there is research on improving data locality [11, 12]. There is also related work in the domain of query optimization, where multiple different queries, optimized for different scenarios, are generated offline and the best query is chosen based on run-time information [13, 14, 15]. Furthermore, there are approaches that perform optimizations based on profiling information recorded at run time [16, 17].

*This research project is partially funded by Oracle Labs.

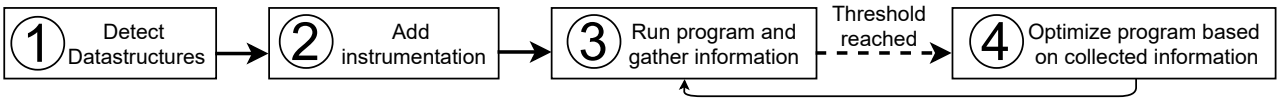

Figure 1. The four steps of the approach: *1) Data Structure Detection 2) Instrumentation 3) Collection 4) Optimization.*

3 Approach

In this paper, we propose an approach to collect information about the processed data at run time and to use it to optimize the code in terms of performance. Our approach consists of four steps, as illustrated in Figure 1:

(1) First, relevant data structures need to be identified. For our approach, data structures are only relevant if they contain lots of elements and are frequently used. These requirements cannot be verified statically, but need to be verified at run time. Hence, this information needs to be gathered during execution with the actual data.

(2) The dynamic runtime inserts additional code that collects information about potentially relevant data structures. The instrumented operations include read and write accesses, to count usages, to track the elements stored in data structures as well as to accumulate statistics about those elements. These statistics are later used to perform optimizations, as described in (4).

(3) The instrumented program is executed in the production environment and information about the processed data structures as well as the access patterns is collected. At some point in time, a data structure might become relevant, because enough elements have been added or the data structure has been used often enough. Reaching this relevance threshold triggers a transition to the next step.

(4) The dynamic runtime uses the collected information to optimize the executed program. Optimizations include program flow optimizations, such as reordering predicates of a conditional statement based on their selectivities, reordering multiple conditional statements or even reordering whole loops containing such predicates. Additionally, the dynamic runtime might adapt the way a data structure is stored in memory to better fit the observed usages. We call this optimization *storage transformation*. This optimization is discussed in more detail in the next paragraph. Further optimizations can be performed in a dynamic compiler based on the collected information, but as these are outside the scope of this work they are not discussed here.

Optimization via Storage Transformation. With storage transformation, we can achieve better data locality for the processed data. This optimization is typically applied to an array of objects, which is transformed to an object of arrays, i.e., a single object is created that contains one array for each distinct property of the original objects. The resulting arrays contain the values of the respective properties of all objects of the original data structure, meaning that these property values are stored in linear memory. Processing data stored in linear memory leads to efficient cache utilization and thus to increased performance. Additionally, linear arrangement of the data enables vectorization [18], improving the performance even further. This transformation is illustrated on an array of `Order` objects in Figure 2. An `Order` has two properties, the customer ID (`custId`) and the total price of the order (`total`). The upper part of the figure depicts the original array, whereas the lower part depicts the transformed data structure.

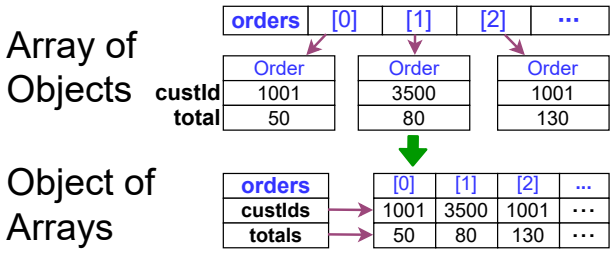

Figure 2. Transforming an array of `Order` objects to an object of arrays.

However, storage transformation is only applicable if certain requirements are fulfilled: First, all objects stored in the array need to have the same shape. Hence, the types of the objects in the array need to be tracked and verified at run time. Furthermore, the transformation only has a noticeable effect if the array is used frequently, which we approximate with access counters.

Apart from the cost of the transformation itself, the downside of the transformation is that writing elements into the transformed array is expensive. When adding an object to the transformed array, each property value has to be written into the respective property array. Hence, the cost of element writes scales with the number of object properties. To tackle this problem, we track the usage of the original array and only perform the transformation if the array is primarily read. Furthermore, we track the usage of the array after transformation, such that if the access patterns change and more write operations occur, we can revert the transformation to prevent excessive overhead for such workloads.

Implementation. The presented approach will be integrated into GraalVM, a high-performance polyglot virtual machine [19]. GraalVM contains the Graal Compiler [20,

21], a high-performance JIT compiler, and the Truffle framework [22], that enables polyglot execution of programs written in multiple languages. Currently, we work on a prototypical implementation of the presented approach for *SimpleLanguage* [23], a Truffle research language exhibiting characteristics of a dynamically-typed, object-oriented language. Our prototype currently supports collecting metrics of arrays and enables storage transformation on arrays of objects.

Evaluation Methodology. Our current prototypical implementation is based on *SimpleLanguage* for which no relevant benchmarks exist. Hence, we use hand-crafted microbenchmarks that focus on operations on large, in-memory datasets of similar objects (akin to in-memory query processing, in-memory databases, etc.). Preliminary results indicate that data-heavy workloads, primarily composed of read accesses, can be sped up by up to 30% using our prototype. In our microbenchmarks, the data analysis causes an overhead of 5% to 15% depending on the specific workload. However, the collected information enables us to perform aggressive optimizations in the dynamic compiler, which more than compensate for the analysis costs. The compiler optimizations are part of the concurrently developed project. While we plan to expand our microbenchmark suite, we also plan to evaluate our approach with other data-centric open-source benchmarks that are supported by Truffle languages.

4 Conclusion

The approach presented in this paper aims at narrowing the performance gap typically seen when executing heavy data-processing workloads in dynamic runtimes compared to traditional database systems. To achieve this goal, the runtime is extended to automatically collect information about the processed data at run time in the production environment. By doing so, the runtime can use this information about the actually processed data to perform specific optimizations and can thus significantly improve the performance of programs.

References

[1] S. F. Oliveira, K. Fürlinger, and D. Kranzlmüller. 2012. Trends in computation, communication and storage and the consequences for data-intensive science. In IEEE. DOI: 10.1109/HPCC.2012.83.

[2] D. Das et al. 2015. Query optimization in oracle 12c database in-memory. *Proc. VLDB Endow.* DOI: 10.14778/2824032.2824074.

[3] P. G. Selinger et al. 1979. Access path selection in a relational database management system. In Association for Computing Machinery. DOI: 10.1145/582095.582099.

[4] M. Jarke and J. Koch. 1984. Query optimization in database systems. *ACM Comput. Surv.* DOI: 10.1145/356924.356928.

[5] R. Ahmed et al. 2006. Cost-based query transformation in oracle. In VLDB Endowment.

[6] D. Abadi, S. Madden, and M. Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In Association for Computing Machinery. DOI: 10.1145/1142473.1142548.

[7] A. Dwivedi, C. Lamba, and S. Shukla. 2012. Performance analysis of column oriented database vs row oriented database. *International Journal of Computer Applications.* DOI: 10.5120/7841-1050.

[8] G. Graefe and H. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In Association for Computing Machinery. DOI: 10.1145/1739041.1739087.

[9] S. Das et al. 2019. Automatically indexing millions of databases in microsoft azure SQL database. In Association for Computing Machinery. DOI: 10.1145/3299869.3314035.

[10] T. Suganuma et al. 2001. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.* DOI: 10.1145/504311.504296.

[11] C. Ding and K. Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. *SIGPLAN Not.* DOI: 10.1145/301631.301670.

[12] M. M. Strout, L. Carter, and J. Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. In Association for Computing Machinery. DOI: 10.1145/781131.781142.

[13] R. L. Cole and G. Graefe. 1994. Optimization of dynamic query evaluation plans. In Association for Computing Machinery. DOI: 10.1145/191839.191872.

[14] Y. E. Ioannidis et al. 1997. Parametric query optimization. *The VLDB Journal.* DOI: 10.1007/s007780050037.

[15] W. Zhang et al. 2021. Adaptive code generation for data-intensive analytics. *Proc. VLDB Endow.* DOI: 10.14778/3447689.3447697.

[16] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. 1991. Using profile information to assist classic code optimizations. *Software: Practice and Experience.* DOI: 10.1002/spe.4380211204.

[17] T. A. Khan et al. 2021. DMon: efficient detection and correction of data locality problems using selective profiling. In USENIX Association. ISBN: 978-1-939133-22-9.

[18] G. M. Duboscq. 2016. *Combining speculative optimizations with flexible scheduling of side-effects.* PhD thesis. Linz, April 2016.

[19] T. Würthinger et al. 2013. One VM to rule them all. In Association for Computing Machinery. DOI: 10.1145/2509578.2509581.

[20] L. Stadler, T. Würthinger, and H. Mössenböck. 2014. Partial escape analysis and scalar replacement for java. In Association for Computing Machinery. DOI: 10.1145/2581122.2544157.

[21] D. Leopoldseder et al. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In ACM Press. DOI: 10.1145/3168811.

[22] C. Wimmer and T. Würthinger. 2012. Truffle: a self-optimizing runtime system. In Association for Computing Machinery. DOI: 10.1145/2384716.2384723.

[23] Oracle. [n. d.] GraalVM - introduction to SimpleLanguage. GitHub. Retrieved 07/05/2021 from https://www.graalvm.org/graalvm-as-a-platform/implement-language/.