Portable Intermediate Representation for Efficient Big Data Analytics

Giannis Tzouros, Michail Tsenos, Vana Kalogeraki

Portable Intermediate Representation for efficient Big Data Analytics

Giannis Tzouros, Michail Tsenos and Vana Kalogeraki

Department of Informatics,
Athens University of Economics and Business,
Athens, Greece
tzouros@aueb.gr, tsemike@aueb.gr, vana@aueb.gr

Abstract. To process big data, applications have been utilizing data processing libraries over the last years, which are however not optimized to work together for efficient processing. Intermediate Representations (IR) have been introduced for unifying essential functions into an abstract interface that supports cross-optimization between applications. Still, the efficiency of an IR depends on the architecture and the tools required for compilation and execution. In this paper, we present a first glance at a framework that provides an IR by creating containers with executable code from structures of data analytics functions, described in an input grammar. These containers process data in query lists and they can be executed either standalone or integrated with other big data analytics applications without the need to compile the entire framework.

1 Introduction

Over the last ten years, massive amounts of information is generated constantly by social media, digital stores, weather stations, traffic sensors, e-commerce, healthcare, smart cities etc. The data created by the aforementioned data domains can be transferred for storage and further processing.

To manage big data, modern applications deploy libraries and algorithms for various big data computations, including aggregates, top-K results, nearest neighbor machine learning, etc. On the other hand, the evolution of applications introduces new types of data that require combinations of existing libraries and/or new libraries, which may lead to performance and efficiency challenges when executing on CPU resources. To deal with these issues, compilers utilize Intermediate Representation (IR) to provide universal computing functions and allow cross optimizations between different libraries and algorithms. [1]. An Intermediate Representation provides an abstraction for otherwise incompatible libraries, which hides details about the target execution platform and expresses data tasks under a unique interface. However, the deployment of Intermediate Representations can be complicated depending on the tools required and the scope of a framework. This problem leads to adaptability issues regarding deploying the representations on diverse environments and portability issues due to potential file size of the source code of each representation.

To address those problems, we present a framework that provides an Intermediate Representation created in Java, which takes as input a context-free grammar with computing function structures and creates portable containers via an extended SableCC compiler [1] that execute the functions described in the grammar. The containers are easily deployable over multiple environments and can be deployed easily on multiple data processing frameworks, potentially improving their compilation and execution times, but also on multi-cluster serverless environments, supporting parallel execution, optimizing resource usage.

2 Design and Challenges

The objective of our framework's intermediate representation is to compile computational functions from different libraries described in a hand-made grammar into a unified abstraction and convert them into portable containers. Our framework's design goals are to provide support for different high-level programming languages and multiple data analytics applications, compatibility with software optimizations for efficient execution and portability and autonomy support for containers so that they can be used without recompiling the main framework.

2.1 Design objectives

More specifically, there are the following three design objectives: First, the containers must include an operational code which can be executed in multiple networking environments, including serverless environments, and must support standalone usage by the client or a variety of applications. Secondly, our framework must ensure that the output data of a container will be ready for use with other containers without any extra changes, in order to optimize execution and reduced processing times between operations. Finally, big data functions inside a container must be available for execution as many times as possible without having to build the entire code of the framework. The containers must be created by the Intermediate Representation only once, so that they can operate as standalone executables on multiple environments.

3 System Architecture

Our Intermediate Representation Container framework consists of the following components: The Intermediate Representation which captures the structures of the functions described in a grammar, the runtime environment which implements the functions captured in the IR into abstraction instances, the SableCC backend compiler which constructs containers based on the function instances made by the compiler and the portable containers deployed in multiple data environments and execute computations on input queries. The architecture and the synergy between the components of the framework is shown on Fig. 1.

[1] https://sablecc.org/

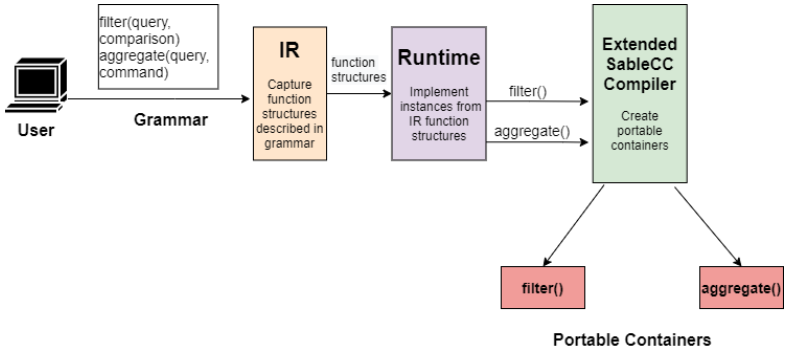

Fig. 1: The architecture of our framework: The IR that captures the function structures of an input grammar, the runtime environment that generates executable code from the IR instances and the extended SableCC compiler which generates serverless portable containers with the generated code.

3.1 Intermediate Representation Model

Our framework creates an IR which collects the most essential functions from different applications and programming languages described in a grammar and integrate them into a single concept, containing universal functions, ready to use in various big data environments. The functions within our framework's IR must contain well structured semantics in order to be later on converted as container executables via the backend compiler.

To convert data analytics functions, our framework's Intermediate Representation takes as input a grammar created by the user, which includes names and structures for these functions.

Data Model The grammar contains two categories of data types: **Scalars**, which are data objects with finite values (int, long, double etc.), and **Lists**, which accommodate multiple values of one or more Scalar types into a single data object (String of multiple char values, dynamic array of numeric values, strings or key-value pairs). The data types are used as parameters for the functions described in the grammar. We chose these data types because they are common in the majority of functions and operations throughout most popular programming languages.

Functions Based on the input grammar, our framework's Intermediate Representation utilizes operators that describe big data analytics functions. All functions take as input a query list with multiple entries, where each entry contains one or more Scalar values either as a key-pair or as a CSV line with more than 2 values. For certain functions, the user or device must insert one or more Scalar values as parameters required by the appropriate function. Currently, our IR supports the following big data functions:

filter(query, comparison): The operation takes as input a query list and a string that describes the filtering requirement of a variable compared to a numeric value and returns a new query list that contains any of the previous query entries that satisfy the entered requirement.

aggregate(query, command): Outputs a computational result for a variable's values within a query list, depending on the value of the string *command*. The *command* string supports the following entries: average, count, sum, median, standard deviation, minimum, maximum, mode and range.

valueScore(query): Outputs the number of occurrence of a value for every entry in the input query list. Namely, how many times each value appears through the entire query.

topKValue(query, k): Outputs the top k entries from the input query in a descending order [2] based on the value of a variable

topKScore(query, k): Similar to *valueScore*, this operation outputs the top k entries from the input query list in a descending order based on the frequency a value occurs.

nearestNeighbor(query, x, y, k, g1, g2): This operation implements the most common nearest neighbor technique: the k-Nearest Neighbor classification algorithm [3]. To classify a target object with coordinates x and y, the function computes the distance (Euclidean or Hamming) between the object and every entry in the query list that belongs either of the groups *g1* or *g2*. Next, the function sorts all computed distances in ascending order, from the entry closest to the object. Finally, the function gathers the top k entries from the list and, depending on the group the majority these entries are, the object is classified to the winning group.

3.2 Runtime Environment

Our framework supports a runtime environment to generate code for function structures captured by the Intermediate Representation. The runtime is developed in Java and takes the main ideas of the cached function: input, computations and output/result and recreates the main operations of the function in Java.

3.3 Backend compiler

Our framework uses an extended version of the SableCC backend compiler. Due to the fact that SableCC compiles grammars only for lexical, syntax and semantic purposes, we enhanced its functions by converting the executable code created by the runtime for every function in the Intermediate Representation into JAR executable files and, next, enclosed the JARs into portable containers which can run in multiple environments. These containers execute their included code when they take a query list and any required parameters as input, as indicated by the function's structure in IR.

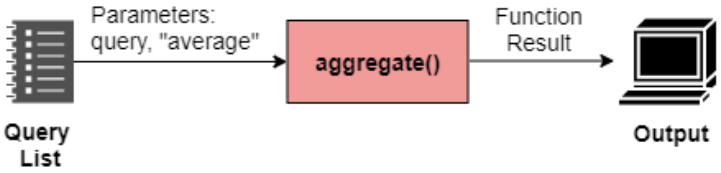

Fig. 2: Architecture of a serverless container that describes its functionality as well as the input data and the output target.

3.4 Serverless Containers

The containers, created from the backend compiler described above, are able to run on either single computers, server clusters or as serverless functions in a cloud provider such as IBM Cloud Functions, Amazon AWS, or in a private cloud with OpenFaas [2]. The container packages up application code and all its dependencies so the application can run in any environment. A container can be replicated and each replica can be referenced as an active instance of the application. In Fig. 2 you can see the execution process of a single container created by our compiler. By choosing the Serverless computing model[4], we have a lower operational and deployment cost due to its unique pricing policy based on a *pay-as-you-use* model. The cloud provider allocates instances on demand, so the number of active instances can be selected either by the application user, or can be adapted dynamically according to the number of requests. During periods of high load the number of active instances is adapted automatically in order to compensate the increased traffic and during extended periods of inactivity the number of active instances can be decreased to zero and release any resource that where allocated.

4 Related Work

Our framework's objective is to unify Intermediate Representation, code compilation and distributed computing into one abstraction, providing Intermediate Representation to gather essential functions from different libraries and languages into one abstraction layer, accessible by any system regardless of its network environment or architecture. To accomplish that objective, our work must provide services and operations that extend or improve on techniques from existing IR implementations. More specifically, LLVM [5] and OpenCL [6] implement low-level language independent Intermediate Representation for uniting and compiling code portions written in different libraries or languages, whereas Weld [7] improves upon the previous works by making the IR compatible with parallel distributed systems using loop fusion and loop tiling. For our framework, we need to gather the structures of the most essential functions from widely used

[2] https://github.com/openfaas/faasd

languages and libraries and describe them in a single grammar file from which our work will construct its Intermediate Representation.

Certain IR frameworks focus only on certain languages or systems, like LINQ [8] which provides IR exclusively for .NET framework systems and PinaVM [9] which utilizes IR for verification of programs written in SystemC [10], a C++ library focused on modeling of systems at different levels of abstractions ranging from functional description to cycle-accurate modeling.. Our framework initially provides IR for queries written in Python, but in the future we can add support for other widely used languages, such as Java, C++ and JavaScript. Also, other frameworks utilize IR for different applications i.e. Halide [11] uses IR to improve image processing, the approach of Vatavu et al. [12] uses IR for detection of 3D models in a traffic road and IF [13] deploys IR on Specification and Description Language (SDL) telecommunication systems. Our framework's IR creates containers that are deployable with a wide range of applications through multiple data environments, including the applications mentioned above.

There are several implementations of runtime systems that generate code for function structures, such as HyPer [14] and LegoBase [15]. These systems, however, are restricted to the relational model and are often difficult to implement in different systems because they need to generate imperative code directly from multiple operators. Our framework contains operators written in Java for constructing code for a series of supported functions.

Container deployment is a well known technique used by implementations like Docker [16], Kubernetes [17] and Firecracker [18] for optimizing code execution on distributed systems. Faasd is an OpenFaas fork, which offers a lightweight portable FaaS engine which can run anywhere, from AWS to small low powered computers, such as Raspberry Pi. In our framework we generate images that are compatible with the Containerd runtime. In this way the user can easily choose the environment that he wants to deploy the function which can scale from its home desktop to fully distributed over Docker Swarm or Kubernetes.

5 Conclusion

In this paper we have presented a framework that utilizes intermediate representation for providing a unique abstraction for functions from multiple libraries or languages and creates portable containers based on these functions, which can be executed standalone by users or applications on multiple environments, without requiring to compile the framework's entire code each time.

Acknowledgements This research has been supported by the European Union through the H2020 952215 TAILOR project.

References

[1] J. Zhao et al. "Formalizing the LLVM intermediate representation for verified program transformations". In: *Proceedings of the 39th annual ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages. 2012, pp. 427–440.

[2] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. "Top-k query processing in uncertain databases". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 896–905.

[3] G. Guo et al. "KNN model-based approach in classification". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2003, pp. 986–996.

[4] X. C. Lin and et al. "Serverless Boom or Bust? An Analysis of Economic Incentives". In: *USENIX*. 2020.

[5] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[6] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.

[7] S. Palkar et al. "Weld: A common runtime for high performance data analytics". In: *Conference on Innovative Data Systems Research (CIDR)*. 2017, p. 45.

[8] E. Meijer, B. Beckman, and G. Bierman. "Linq: reconciling object, relations and xml in the. net framework". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006, pp. 706–706.

[9] K. Marquet and M. Moy. "PinaVM: a SystemC front-end based on an executable intermediate representation". In: *Proceedings of the tenth ACM international conference on Embedded software*. 2010, pp. 79–88.

[10] D. C. Black et al. *SystemC: From the ground up*. Vol. 71. Springer Science & Business Media, 2009.

[11] J. Ragan-Kelley et al. "Halide: Decoupling algorithms from schedules for high-performance image processing". In: *Communications of the ACM* 61.1 (2017), pp. 106–115.

[12] A. Vatavu and S. Nedevschi. "Real-time modeling of dynamic environments in traffic scenarios using a stereo-vision system". In: *2012 15th International IEEE Conference on Intelligent Transportation Systems*. IEEE. 2012, pp. 722–727.

[13] M. Bozga et al. "IF: An intermediate representation for SDL and its applications". In: *SDL'99*. Elsevier, 1999, pp. 423–440.

[14] T. Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[15] Y. Klonatos et al. "Building efficient query engines in a high-level language". In: *Proceedings of the VLDB Endowment* 7.10 (2014), pp. 853–864.

[16] *Docker*. `https://www.docker.org`.

[17] E. A. Brewer. "Kubernetes and the path to cloud native". In: *Proceedings of the sixth ACM symposium on cloud computing*. 2015, pp. 167–167.

[18] A. Agache et al. “Firecracker: Lightweight virtualization for serverless applications”. In: *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*. 2020, pp. 419–434.