Open Access

Open Journal of Cloud Computing (OJCC)
Volume 6, Issue 1, 2019

http://www.ronpub.com/ojcc
ISSN 2199-1987

Code Generation for Big Data Processing in the Web using WebAssembly

Sven Groppe, Niklas Reimer

Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany,
groppe@ifis.uni-luebeck.de, Niklas.Reimer@student.uni-luebeck.de

Abstract

Traditional clusters for cloud computing are quite hard to configure and setup, and the number of cluster nodes is limited by the available hardware in the cluster. We hence envision the concept of a Browser Cloud: One just has to visit with his/her web browser a certain webpage in order to connect his/her computer to the Browser Cloud. In this way the setup of the Browser Cloud is much easier than those of traditional clouds. Furthermore, the Browser Cloud has a much larger number of potential nodes, as any computer running a browser may connect to and be integrated in the Browser Cloud. New challenges arise when setting up a cloud by web browsers: Data is processed within the browser, which requires to use the technologies offered by the browser for this purpose. The typically used JavaScript runtime environment may be too slow, because JavaScript is an interpreted language. Hence we investigate the possibilities for computing the work-intensive part of the query processing inside a virtual machine of the web browser. The technology WebAssemby for virtual machines is recently supported by all major browsers and promises high speedups in comparison with JavaScript. Recent approaches to efficient Big Data processing generate code for the data processing steps of queries. To run the generated code in a WebAssembly virtual machine, an online compiler is needed to generate the WebAssembly bytecode from the generated code. Hence our main contribution is an online compiler to WebAssembly bytecode especially developed to run in the web browser and for Big Data processing based on code generation of the processing steps. In our experiments, the runtimes of Big Data processing using JavaScript is compared with running WebAssembly technologies in the major web browsers.

Type of Paper and Keywords

Regular Research paper: *Browser Cloud, Big Data, Browser Virtual Machine, WebAssembly, JavaScript*

1 Introduction

Complex data processing tasks are typically splitted into basic operations, which form an operator graph. In the traditional way taken by databases each operator offers the same interface, the *iterator interface*, such that the operators can be orthogonally composed together in the operator graph [10]. The iterator interface offers especially a method to retrieve the next computed result, such that first results are determined before the whole data has been processed and materialization of intermediate results to external storage like hard disks or SSDs is often avoided.

However, if all operators implement the same interface and can be dynamically composed in the operator graph, the method calls need to be virtual using indirect addresses of the method implementations, which is the standard approach for calling overridden methods in object-oriented languages. Indeed virtual method calls are slower than direct method calls, where the addresses

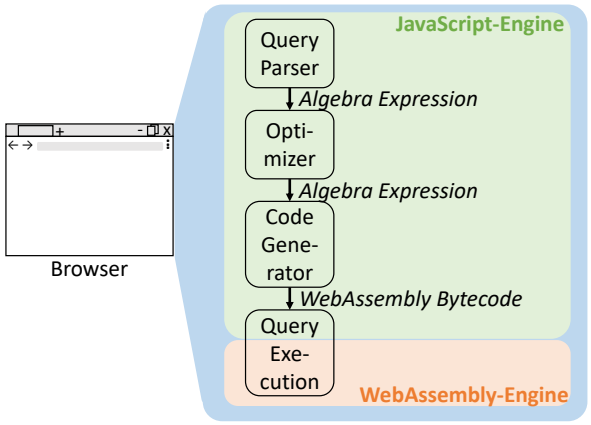


Figure 1: Query processing in the browser via code generation using an online kotlin compiler

of the method implementations are fixed and can be determined at compile-time. For typical applications, the difference in performance is insignificant. However, for big data processing, where millions, billions or more data items need to be processed, methods for data processing need to be also called millions, billions or more times, such that the performance is drastically increased whenever direct addressing is used to call methods. Whenever queries to be processed are retrieved at runtime, direct method calls during data processing can be only achieved if code for the single data processing steps are dynamically composed together, compiled and executed. This approach is called *code generation* [19].

For a long time, the Web environment doesn't seem to be suitable for Big Data processing, because there was no technology to establish a network between the browsers of a website, and browsers didn't offer enough computing and storage resources. The situation changes from browser generation to browser generation by supporting technologies like WebRTC [29] for real-time communication capabilities, and JavaScript [28] and WebAssembly [27] for executing programs. WebAssembly defines bytecode to be run in a virtual machine and promises faster execution times than the interpreted JavaScript language.

Kotlin [15] is currently the rising star among newly developed programming languages with 2.6 times growth in contributors in one year on Github [11]. Furthermore, Kotlin is ideally designed for large-scale data processing [12]. Hence we focus on the use of Kotlin as programming language for formulating the generated code for Big Data processing.

Code generation for Big Data processing requires a compiler that runs directly in the browser (see Figure 1) to start query processing at any browser in the network. Alternative architectures with a server increases latency and communication costs because of introducing additional communication over the network. However, there is a lack of WebAssembly compilers that directly run in the browser. Hence our contribution includes a compiler from a Kotlin subset to WebAssembly bytecode.

Furthermore, we run experiments to validate the performance speedup of WebAssembly bytecode executing Big Data processing tasks in comparison to running generated code in JavaScript.

2 BASICS

While we introduce relevant programming languages in Section 2.1, we describe different types of code generation in Section 2.2 as basics about code generation for query processing. We introduce the related work in Section 2.3.

2.1 Programming Languages

For the implementation we use the programming languages JavaScript (for our online compiler) and Kotlin (for the multiplatform engine), as well as WebAssembly bytecode as compilation target for executing the generated code from the query. For a better overview we introduce them with focus on the special features in this context.

2.1.1 JavaScript

JavaScript [28] is an object-oriented general-purpose interpreter language. It's origins date back to 1995 when JavaScript, which was called LiveScript at that time, has been integrated into the Netscape webbrowser to introduce interactive features for websites.

JavaScript runs a garbage collector in the background, which task is to clean up the memory by identifying and deleting superflous objects and leaving the free space for the operating system. Being an interpreted language, the code isn't compiled beforehand. It is processed in the background when accessing a website. Over the years JavaScript has become a very integral part of the browser engines and the browser developers continously increase their efforts to provide better JavaScript performance. However, JavaScript execution is still outperformed by machine code by a factor of 10-100. Especially games or applications with huge amounts of data suffer a lot from this fact. [5]

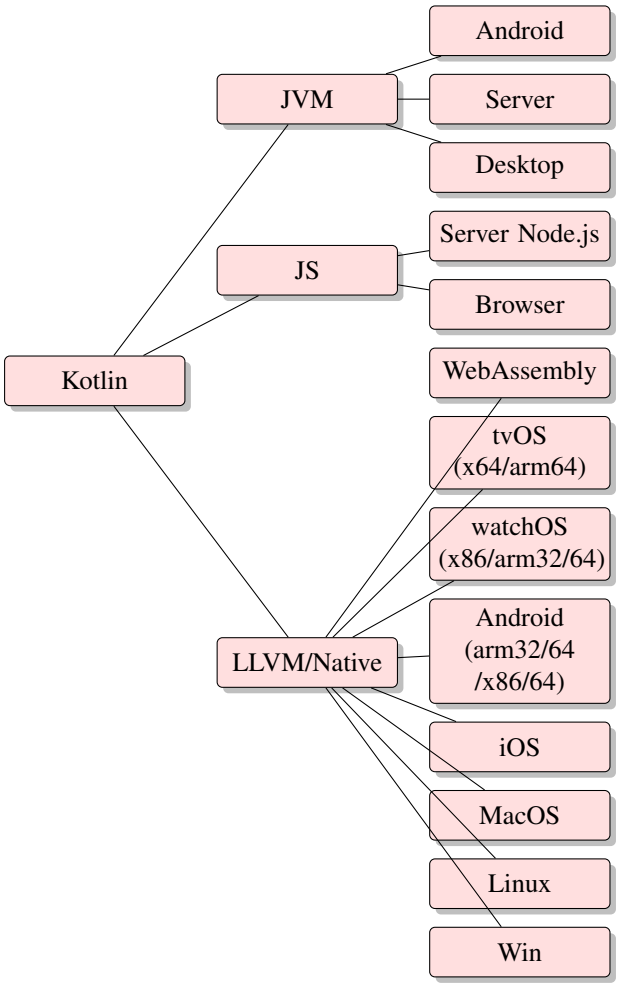


Figure 2: Targets of Kotlin

2.1.2 Kotlin

Kotlin is a general-purpose programming language being developed since 2010 by *Jet Brains s.r.o.*. It's free software as it's source code is published unter the Apache 2 licence. [15]

Kotlin offers some special features including the support for several platforms like JVM, Android, JavaScript and a native execution in one multiplatform project (see Figure 2). Through combination of elements of object-oriented and functional programming as well as readable short-hand notations for often used code patterns, many programs can be expressed in Kotlin using less code than in most other programming languages. Parser generators can be feeded with the context free grammar of the Kotlin syntax to generate very fast parsers of the Kotlin language.

For code generation it's very handy to use a compact, but easy processable programming language. Furthermore, it is planned to develop a multiplatform Big Data analytics engine, which owns a common code basis for Big Data processing over several target platforms. Because of Kotlin's multiplatform feature, this multiplatform Big Data analytics engine [7] is currently being developed in Kotlin. Hence it is very consistent to use Kotlin as programming language for the web compiler for running the generated code for Big Data processing.

2.1.3 WebAssembly

The WebAssembly technology [27] includes a virtual machine that is being executed inside of the web browser. It's instances are stored, controlled and called using JavaScript.

WebAssembly's virtual machine is based on the principle of the stack machine. As the stack is able to store several data types, it's necessary to save additional information about the data type of the elements on the stack. This method prevents that operations are being executed on elements of a different data type.

As a new way to deploy established programming languages to the web browser, WebAssembly has a great potential. Hence there exists already a huge collection of projects that compile various languages to WebAssembly. [3]

The currently available version 1.0 is the very first release that is ready for productive use. It's focused on the four skills *compilation target, fast execution, compactness and linear memory* to run C/C++ code and is called *Minimum Viable Product* (MVP). This is also the first version that has been shipped with major web browsers.

Before executing the bytecode it's subjected for a machine-dependent optimization in the browser. Additionally, a semantically equivalent and more human-readable text representation has been specified that can be used during the development process.

By now the only way to instantiate a WebAssembly module is by running a JavaScript program that calls special functions of a JavaScript API for WebAssembly. For this purpose, the developer creates a module with the compiled bytecode that can be instantiated afterwards. Then the developer specifies the glue that provides information about variables, objects and linear memory that JavaScript and the WebAssembly instance will share. Finally the glue gets assigned to the WebAssembly instance.

2.2 Code Generation

We differ between two main types of code generation:

- *offline code generation* for generating code before the application is compiled, and

- *online code generation* for generating and executing code at runtime of the application.

The online code generation is being used by modern database engines like Apache Spark and Apache Flink to speed up query processing. In comparison to these Big Data frameworks, we developed an online compiler for code generation for Big Data processing in the *Web* instead of staying in the Java virtual machine (JVM). In our experiments, we hence investigate the benefits of running WebAssembly in comparison to JavaScript for Big Data processing in the web browser (see Section 4).

2.2.1 Offline-Code Generation

In general, a set of rules is applied during code generation, where an input is transformed to code for compilation and execution. In the offline variant this process is done before the application is compiled. Indeed typically the generated code is compiled together with the code of the application itself.

This type is often used for (de)serializing objects from and to the memory. Despite to some proprietary formats the serialization [6] often uses some standardized formats like XML and JSON.

2.2.2 Online-Code Generation

Online code generation generates code at *runtime*, which is compiled to machine code (or to bytecode in case of using the JVM) and executed during the same run of the program. In most cases, the generated code is formulated in the language of the application triggering the code generation.

JavaScript comes along with the built-in function *eval()* offering to compile[1] and execute JavaScript code, while other languages like Java require external libraries like the Janino compiler [14] for compiling and executing generated code at runtime.

The generated code is faster as it won't contain any virtual calls or load static values from variables. Especially during processing huge amounts of data it's possible to show [19] that generating code can improve the speed of query processing a lot.

Usually the process of generating, compiling and finally executing code takes more time than executing conventional code once (t_{naive}). However, because the runtime t_{run} of the generated code is probably less, the following equation system calculates how many iterations n are needed to amortize the costs of generating t_{build} and compiling $t_{compile}$ code:

$$t_{build} + t_{compile} + n \cdot t_{run} \leq n \cdot t_{naive}$$

We run some experiments[2] to test if code generation already has some benefits for pure JavaScript execution: We observed an average execution time of $T_{naive} = 0.95\mu s$ for a general vector multiplication (of a vector of size 4) with a constant (see Listing 2). A corresponding code generation approach (see Listing 3) took $t_{build} + t_{compile} = 1ms$ for generating and compiling the code. The generated code (see Listing 4) runs in $t_{run} = 0.02\mu s$, such that after 1075 iterations the costs are amortized for generating and compiling the code.

2.3 Related Work

Brodersen [7] developed a first prototype of a Multiplatform Big Data Analytics Engine, i.e., it's able to run inside the Java Virtual Machine (JVM) and inside a JavaScript engine in the browser (using Kotlin's feature of multiplatform development). The realized way of code generation lacks of a data serialization and therefore hands over iterable objects to the generated code. This is possible due to the fact that the generated code is executed in the same runtime environment, such that it can access all objects and the whole function library of the engine. The engine generates code in two different ways: The JVM variant uses the Janino-Compiler [14] while the JavaScript variant uses the built-in *eval()* function. Experimental results show a significant improvement of the performance when using code generation. Furthermore, Big Data processing via the *eval()* function is much slower in comparison to the Janino compiler.

Neumann's contribution [19] includes a translation from database queries into C++ and assembly code for the virtual machine LLVM that is finally being translated to machine code. Starting from the point that hand-written code, but also generated program code, can be optimized for a very specific query by utilizing the available resources better than traditional database systems like *MySQL* [21] or *MonetDB* [17]. In the experimental results code generation reduces the execution times and the number of mispredicted branches of execution and cache misses that require to load missing data from the memory to the cache of the processor.

A contribution by Nagel et al. [18] deals with in-memory databases. He used the *LINQ* Framework [16] by Microsoft to store and access his data inside of a C# program. Like the contribution in [19] he also used a high-level programming language, in this case

[1] Indeed modern browsers apply Just-In-Time (JIT) compilers like V8 (Chrome/Chromium), SpiderMonkey (Firefox), Chakra (Edge) and JavaScriptCore (Safari) to compile JavaScript to machine code.

[2] Our test system is a Microsoft Surface Pro (2017), Model 1796, running Firefox

C# instead of C++ for the overall control flow of the query processing. However, the operations on the data are generated by a low-level (but close to hardware) language C (instead of the LLVM assembler). The experimental results also showed a huge improvement of the performance and a much more efficient use of the cache inside the CPU.

Horvath et al. [13] deal with the serialization of Java objects in Apache Flink. It turned out that the serialization is often the bottleneck by taking more than 20% of the execution time. By implementing code generation in combination with the Janino compiler Flink generates optimized methods that offer a serialization with a speedup up to factor six. The data format used for the serialization is critical for performance, too, such that the generated methods are optimized and were incompatible to the binary format of the conventional methods.

The contribution [4] of Armbrust et al. describes the implementation of relational schemas for Apache Spark. Here, users provide queries similar to SQL in a domain-specific language. The given queries are optimized by *Catalyst* [8] by first generating an operator tree, which is further optimized. Subsequently an internal compiler directly generates JVM bytecode from the operator tree. The performance for the executed test cases is doubled in experiments in comparison to a traditional implementation in Scala without code generation. This method can also be applied to Python applications with much higher performance speedups up to a factor 12.

Hence there are already some contributions to code generation in database management systems, but to the best of our knowledge it has not been dealt with code generation for query processing in browsers (especially using WebAssembly code). Furthermore, we also introduce the concept of a Browser Cloud in this paper.

3 Multi-Platform Big Data Analytics Engine

In this contribution we deal with code generation for Big Data analytics in the browser environment using WebAssembly. Our contribution is embedded in a vision of a new generation of Big Data Analytics Engines to be introduced in Section 3.1, which work on multiple platforms (see Section 3.1.1) and are also running in a cloud of browsers (see Section 3.1.2). Furthermore, we describe the memory organization of our proposed engine in Section 3.2 and the developed online compiler used for compilation (and execution) of the generated code from queries in Section 3.3.

3.1 Vision

Our vision for a new generation of Big Data Analytics Engines is two-fold: We envision a multiplatform development, such that the engine is available for several platforms with considerable benefits as discussed in Section 3.1.1. We show in [7] the feasibility of a multiplatform engine (for the JVM and browser target platforms). Furthermore, we are of the opinion that the browser environment has advantages in comparison with traditional Clouds (see Section 3.1.2). We elaborate on the Browser Cloud in [26].

3.1.1 Multiplatform versus Single-Platform Big Data Analytics Engine

Multiplatform Big Data Analytics Engines have considerable benefits in comparison to engines running only on a single platform. We identify the following benefits:

Development Efforts: The development of a Multi-Platform Big Data Analytics Engine comes along with initially more efforts in comparison to the variant for a single platform. However, if more platforms are considered, the single platform variant needs to be ported and sometimes even the programming language needs to be changed. In contrast, in the multiplatform variant platform-specific code has already been identified to be modified for the different platforms and platform-independent shared code (being supported by Kotlin) runs in different platforms without modifications.

Reusability: The reusability of code is much larger for the Multiplatform Big Data Analytics Engine, because new platforms can be easier integrated avoiding new developments from scratch.

Learning Curve: As a matter of fact, the configuration of multiplatform projects are more complicated, and the used tools and the whole project structure are more complex. Hence, the learning curve is initially higher for the multiplatform variant in comparison to single platform developments. However, clever applying development patterns and available tools may help to reduce the learning curve.

Performance: Similar performance for the multiplatform in comparison to the single platform variant can be achieved. Kotlin supports platform-specific declarations by marking a class as expected in the shared code, which must be implemented for the different target platforms. With clever usage of this feature, the multiplatform developer doesn't introduce

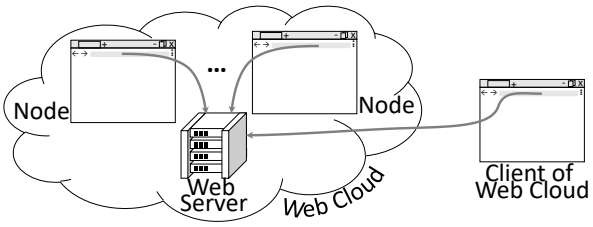


Figure 3: The Browser Cloud

additional overhead in the binaries and may avoid costly virtual calls.

Administration Effort: The Multiplatform Big Data Analytics Engine has the great advantage of shared code between the different platforms. Bugs in the shared code hence need to be fixed and the shared code for new features need to be developed *only once*, and not for each of the platforms like it would be the case for ported single platform engines.

Communication and I/O Interoperability between different Platforms: One might also consider to develop (de)serialization routines for communication and file storage as much as possible as shared code. In this way 100% interoperability of communication and file formats between instances running on different platforms is easier to achieve in comparison to develop several single platform engines.

3.1.2 Browser Cloud

Cloud Computing offers the easy access of computing resources via a network according to the NIST definition of Cloud Computing [24]. Peer-to-peer (P2P) networks can comply with these properties of Cloud Computing [25], even if peer-to-peer networks are unreliable [9]. In these *P2P Clouds* each computer of the network offers its (computing and storage) resources.

Definition of Browser Cloud: A Browser Cloud is a P2P network offering Cloud Computing features, which is built upon browser instances (i.e., the peers are the browser instances).

In Table 1 we compare the Browser Cloud (see Figure 3) with traditional Clouds (e.g., a Hadoop cluster).

The network architecture of a traditional Cloud is typically built after the master/slave principle. If the master node crashes, the whole cluster is affected: Hence there is a single point of failure (SPOF) in principle. The disadvantage of the SPOF is often weakened by running some few standby master nodes taking over the functionality of the master in case of a crash. In contrast, the functionalities of all participating browsers are equal in the Browser Cloud except of the webserver delivering the webpage for adding browsers to the Browser Cloud. Hence an already running Browser Cloud can continue its work after some browsers or even the webserver crash. However, if the webserver crashes, no new browsers can be added to the Browser Cloud, which might be a kind of SPOF with low impact preventing a further scaling.

In traditional Clouds, a computer must have a corresponding runtime environment before the computer can be part of the Cloud. The necessary runtime environment often not only includes the installation of the *Java Runtime Environment*, but also the configuration of a distributed file system (e.g., Hadoop Distributed File System (HDFS)). The installation of the master node requires much more configuration and installation steps. In case of the Browser Cloud, a webserver needs to be setup, which delivers the program for the participating browser instances. Hence the clients only need to run a browser instance and the browser software is included as standard software in most operating system distributions.

The administration of the Browser Cloud is also easier in comparison to a traditional Cloud. In the Browser Cloud the administrator only needs to take care that the webserver remains accessible. In traditional Clouds the administrator also may take over tasks like exchanging hardware (for increasing storage and computing capacities of the clients) and tuning the configuration from time to time based on changing requirements of the software running in the Cloud.

The high requirements of the hardware are advantage and disadvantage of the traditional Cloud at the same time. In fact powerful CPUs and fast read and write accesses improve performance, but also increase the costs for a computer in the Cloud. Costs and setup efforts are hence two issues, which impedes horizontal scaling in the traditional Cloud. In contrast, scaling a Browser Cloud is quite simple: The corresponding website containing the program for integration into the Browser Cloud needs only to be called from another browser instance. Afterwards the resources offered by this browser instance can be utilized in the Browser Cloud. On powerful computers, even several browser instances can be run (in different processes) on the same computer, such that this computer offers several peers to the Browser Cloud.

A disadvantage of the Browser Cloud is the low availability of single peers (but not of the whole Browser Cloud). While a single slave in the traditional Cloud

Table 1: Comparison: Browser Cloud versus traditional Cloud (e.g., Hadoop)

Property	Browser Cloud	trad. Cloud
Central Server (SPOF)	(×)	✓
Mostly homogeneous computers	×	✓
Setup effort	low	high
Administration effort	low	high
Simple horizontal scaling (new network node)	✓	×
Constant number of computers (in typical case)	×	✓
High availability of computers	×	✓
Initiator is part of the network	✓	×
Powerful hardware necessary	×	✓
Fast network connection necessary	×	✓

crashes in seldom cases, leaving of a peer in the Browser Cloud is a typical case (especially for publicly available websites). Hence the resources offered by the Browser Cloud may fluctuate much, but the resources offered in the traditional Cloud remain usually approximately constant.

3.2 Big Data Memory Organization

The memory organization is one of the main factors for the performance of our engine for query and Big data processing in the browser. Hence we describe our approach to the data layout in memory here. The memory organization especially determines the number of cache misses during program execution for transferring the data from memory over the caches to the CPU, and hence overall how efficient the data can be accessed.

For accessing the column *ID*, all attributes of a person in Table 2 must be loaded or memory cells must be skipped. A more efficient approach is the column-oriented model like in Table 3, where all values of a column are close to each other. Whenever the values of a column are accessed, then less cache misses occur for the column-oriented model, as more values of the column are directly loaded to and fit into the cache [1].

Additionally, a simple raw binary data buffer (i.e., JavaScript ArrayBuffer objects) instead of object-oriented programming features is used. This allows serialization without any overhead of metadata as stored in JavaScript objects. The data format *Parquet* [2]

Table 2: Example for row-oriented storing of data

$memory \rightarrow$

ID	lastname	firstname	birthday
1	Mouse	Mickey	1928-11-18
2	Duck	Daisy	1940-06-07
3	Duck	Donald	1934-06-09

Table 3: Data from Table 2 in column oriented format

$memory \rightarrow$

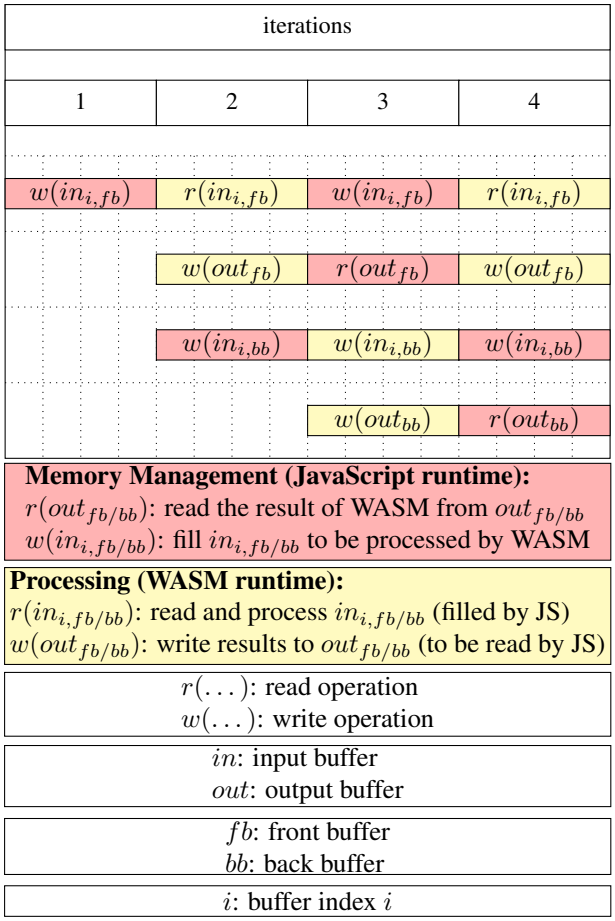
ID	1	2	3
lastname	Mouse	Duck	Duck
firstname	Mickey	Daisy	Donald
birthday	1928-11-18	1940-06-07	1934-06-09

developed by the Apache Software Foundation and used by Twitter and Yahoo follows a similar approach to column-oriented storage. Another advantage is that all values of a column have the same data type, such that this data type only needs to be stored once at the beginning of the column. Furthermore, skipping whole columns without processing is easy for the column-oriented model, whereas each column value needs to be skipped for the row-oriented model.

For Big Data, the whole data to be processed cannot be hold in main memory, because it's typically too large. Hence the data must be block-wise processed. Each of the block must be deserialized before the data block is processed and the result of the data processing must be serialized. Whenever the old block has been processed and a new block must be read in, or the block holding the result is full and must be stored or processed in another way, then the data processing is blocked. To avoid blocking we use a double buffer approach adapted from computer graphics [23]. It uses two input buffers for each input relation and two output buffers for the result, where I/O transfers and data processing are processed in parallel with alternating roles of the buffers after a buffer has been emptied or filled. I/O operations in the browser are not directly supported by WebAssembly and are implemented in our Big Data application by applying JavaScript ArrayBuffer objects in the JavaScript runtime. Objects for the input buffer for the WebAssembly code are filled using JavaScript evaluations and afterwards read and processed by WebAssembly code. WebAssembly code then fills the output buffer (in form of a JavaScript ArrayBuffer object), which is read from the JavaScript runtime for further processing.

We visualize the double buffer approach in Figure 4, where processing steps of the WebAssembly code (i.e., reading the input buffer and writing the output buffer)

Figure 4: Double buffering of input and output in Big Data applications



are marked in yellow and the memory management of the JavaScript runtime (i.e., filling the input buffer and reading the output buffer) in red. For example, in step 3, the JavaScript runtime fills the buffers $in_{i,fb}$ with new data and also reads the results of the last iterations from out_{fb}. At the same in the WebAssembly runtime, input buffer $in_{i,bb}$ is being processed while writing the results into out_{bb}. After each of those iterations the frontbuffers fb swap the roles with the backbuffers bb. In this way data is being read in, processed and read out at all times.

3.3 Requirements of the Online Compiler

The online compiler is used to compile the generated code from the query and hence a key component of our engine for query and Big data processing in the browser. The compiler must support a programming language that includes important constructs of query processing. As the compiler challenges the direct execution of pre-compiled code the compiler should be very fast but at the same time shouldn't bloat the application.

3.3.1 Language Design

The design of the language is a very essential contribution of our work. We get inspired by the Kotlin language, because its programs are concise and quite efficient to compile, and decided to support a large fragment of Kotlin including important constructs for data processing. For that matter, we introduce supported language constructs and the required data types, and describe the way they work. If possible we also refer to and describe a corresponding construct in WebAssembly.

- **Booleans** are represented by $i32$ in WebAssembly as 32-bit integer.
- **Integers:** Depending on the size WebAssembly offers $i32$ and $i64$ for 32- and 64-bit integers.
- **Floating-point numbers:** Depending on the size and desired precision WebAssembly offers $f32$ and $f64$ to store them with 32/64-bit length.

Strings don't have an equivalent in WebAssembly. During data processing, they are only stored in linear memory for in- and output of data. A string s is interpreted as an array of the length $|s|$ of the string and $|s|$ bytes are stored in linear memory at a certain offset with the possibility of a byte-wise access. There is no need to implement them as a special data type class, because operations on strings like the comparison of two strings are mapped to the byte-wise operations (here comparison) of their byte arrays.

Based on our experiences with the Semantic Web database LUPOSDATE [10] and our Multiplatform Big Data Analytics Engine [7], we collected the following features of the programming language necessary to fulfill the fundamental requirements of data processing.

- **Functions and function calls** provide the interface for JavaScript to start WebAssembly applications. While code processed by the Janino compiler in Java or JavaScript's *eval()* function may contain single instructions, the smallest unit of WebAssembly is a function.
- **Variables** are used as a temporary storage inside of functions.
- **Global variables** are not tied to a single function. All functions inside a module are able to use them. Furthermore, they still exist after the function call, such that they store results of additions or counters without returning them back to JavaScript.
- **Static values** are supposed to be used for values that never change. During the compilation progress they are already replaced.
- **Arithmetic operations** like $+, -, \cdot, \div$ are signed operations like in Kotlin for the standard primitive

number types[3].

- **Relational operations**
  - Our compiler supports pure relational operators like $<, \leq, >$ and $\geq$.
  - The operators $=$ and $\neq$ can be used for numbers as well as for Booleans.
  - The logical operators $\wedge$ and $\vee$.
- **Control structures**
  - **Program branches** like `if`/`else`-Constructs
  - **Loops**
    * `do`...`while`-loops
    * `while`-loops
    * `for`-loops
- **Memory access** is realized via `memory[index]` to read or write a part of the memory, which is e.g. necessary when reading the value of a variable or when a new value is assigned to a variable.
- **Terminal outputs** are supported mainly for debugging purposes. As WebAssembly can't access the webbrowser directly access to JavaScript's *console.log*-function is provided by sharing it using glue.
- **Braces** modify the execution order of the operators and curly braces define code blocks.

There are some features that are offered by Kotlin or other programming languages that our implementation lacks of. This is intentional as our goal is to create a fast environment for database operations rather than the support of a complete programming language, which slows down the compilation. This approach is also supported by Nagel [18] and Neumann [19].

We developed a context-free grammar as formal representation of the language. Context free grammars can be transformed to fast parsers using e.g. ANTLR4 [22], which also supports JavaScript as target language of the parser to be run in the browser. Furthermore, these parsers generate an abstract syntax tree of the parsed program. For our online compiler we hence developed the transformation from the abstract syntax tree to WebAssembly bytecode, which is - besides some technical issues of WebAssembly - overall a straightforward process.

4 Experimental Evaluation

We introduce our experimental environment in Section 4.1, the programs, queries and data used in the experiments in Section 4.2 and an analysis of the experimental results in Section 4.3.

[3] Kotlin 1.3 supports unsigned numbers (Byte, Short, Int, Long) and unsigned operations on them, which are marked as experimental.

Table 4: Versions of the software used during the evaluation

Microsoft Surface Pro 2017 (Model 1796)

Application	installed version
Microsoft Windows 10	1809 (Build 17763.195)
Mozilla Firefox	63.0.3
Google Chrome	71.0.3578.98
Java Runtime Environment	8u192
ANTLR	4.7.1
ANTLR Runtime	4.7.1

4.1 Experimental Environment

Table 4 lists the main properties of our experimental environment.

The used device has a mechanism to adjust the clock speed of the processor: It's lowered in idle and runs at full speed under load. To prevent any impact by this feature it had been disabled.

4.2 Programs, Queries and Data

We describe our investigated programs, data and queries used in our experiments for showing the benefits of an online compiler to WebAssembly.

- **Fibonacci numbers** are an infinite series of non-negative integers, that were named after die mathematician Leonardo Fibonacci. The individual members of the series grow with increasing n very fast and are built by adding up it's two predecessors:

$$fib_{n\in\mathbb{N}} = \begin{cases} fib_{n-1} + fib_{n-2} & \text{if } n > 1 \\ n & \text{if } n \leq 1 \end{cases}$$

  In computer science this representation can be easily implemented using a recursive function. However, the recursive variant generates a lot of function calls and additions, such that it's a commonly used benchmark for the performance of a system qualifying the recursive fibonacci program for our experiments as well.
  Despite it's possible to run an iterative function that saves the two predecessors and offers a much more efficient calculation using a loop. We used this variant to compare the performance of loops between JavaScript and WebAssembly.
- **TPC-H Benchmark:** The speed of a database system highly depends on the complexity of the queries and on the size of the data. For this reason various use cases have their special, standardized benchmarks. To measure the speed of transaction systems the *Transaction Processing Performance Council (TPC)* [30] provides data sets and queries for some scenarios.

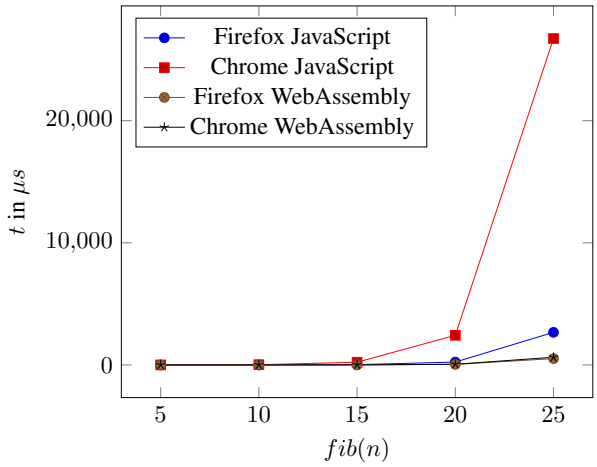


Figure 5: Comparing recursive fibonacci calculations

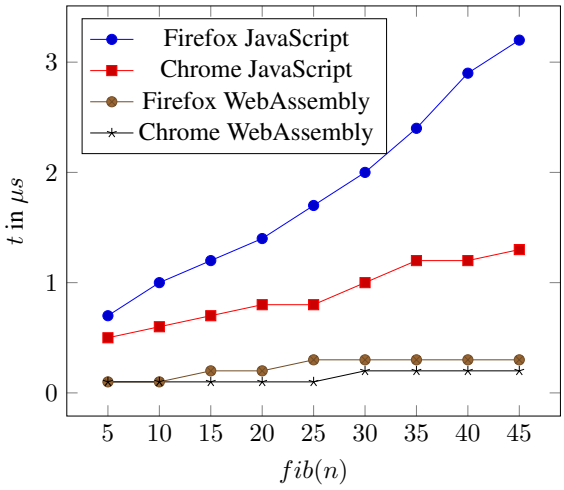


Figure 6: Comparing iterative fibonacci calculations

The TPC defined a data set with a fixed size of 8 tables for the scenario of decision support. A typical benchmark query of TPC-H is the query 6 (see Listing 1).

```
SELECT
 sum(l.extendedprice * l.discount) as revenue
FROM
 lineitem l
WHERE
 l.shipdate >= date '1994-01-01'
 AND l.shipdate <
     date '1994-01-01' + interval '1' year
 AND l.discount between 0.06 - 0.01 AND 0.06 + 0.01
 AND l.quantity < 24;
```

Listing 1: SQL code of TPC-H Query 6

4.3 Analysis

Our experimental results show a high execution performance of WebAssembly bytecode for data processing tasks. For two independent functions, three database operators as well as a query of a database benchmark a higher performance is achieved running WebAssembly bytecode instead of comparable JavaScript code. The performance differences grow with increasing size of the input data. Hence WebAssembly has the required expressive power to execute these processing steps and is even able to accelerate them.

By considering the results of Figure 5 for calculating the fibonacci numbers we observe a massive growth of the execution times with increasing input data size. Especially for the JavaScript function running inside the Google Chrome browser the execution times grow very fast while Mozilla Firefox runs the same code much faster. Still both browsers have significantly lower executions times when using WebAssembly instead of JavaScript.

As the runtime complexity of the recursive fibonacci calculations is $\mathcal{O}(1.62^n)$ the required time for execution grows very fast [20]. By looking at the pretty similar results when using WebAssembly in Firefox and Chrome we can conclude that the used features of the language, in this case recursive function calls, additions and *if/else* control structures, perform a lot faster in this combination than correspondig pendants in JavaScript do. These experiments also indicate that the JavaScript engines inside the browsers offer different performances based on internal differences of their implementations.

Despite to the recursive variant shown before, the amount of time needed grows much slower and in a linear way for the iterative fibonacci calculation (see Figure 6). We see that the JavaScript function running in Firefox is the slowest, followed by the JavaScript one in Chrome. Nevertheless the WebAssembly equivalents seem to have almost equal execution times which grows slowly by increasing the ficonacci number n that will be calculated.

The shallow rising of the curve can be explained through the linear runtime of $\mathcal{O}(n)$ needed for the iterative calculation. As we use a *for* loop instead of a recursive call we can also conclude that these are also faster executed inside the WebAssembly VM. We also see very similar execution times in the WebAssembly variants, while the JavaScript function differs between the two browsers due to their different implementations of the interpreter: Chrome seems to be much faster here and the difference grows with rising n.

In Figure 7 we see that the runtime grows linear for the projection operator depending on the size of the input. All measured variants have that in common. Also we observe much slower execution runs for both JavaScript implementations, being slower by the factor

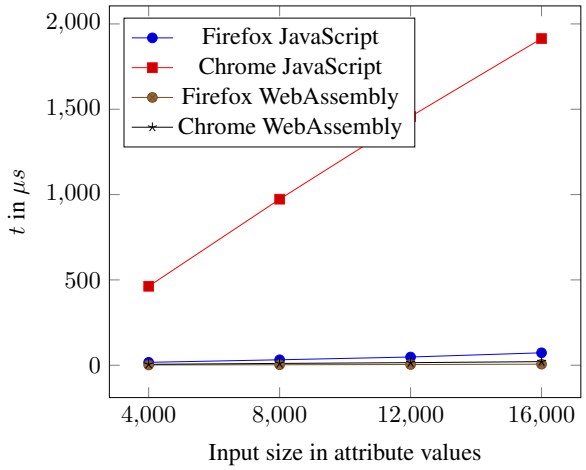


Figure 7: Comparing the projection of a single relation

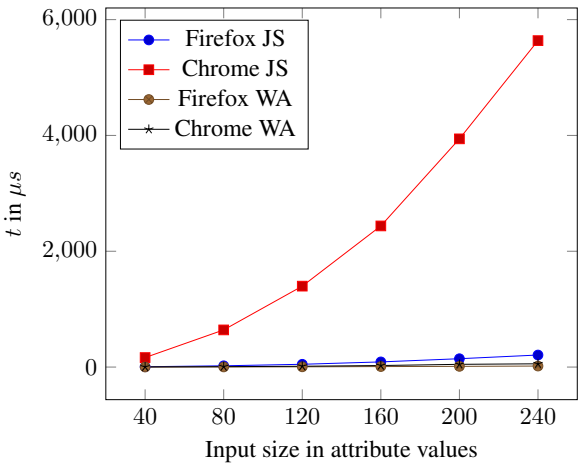


Figure 9: Comparing the execution times to generate a cartesian product

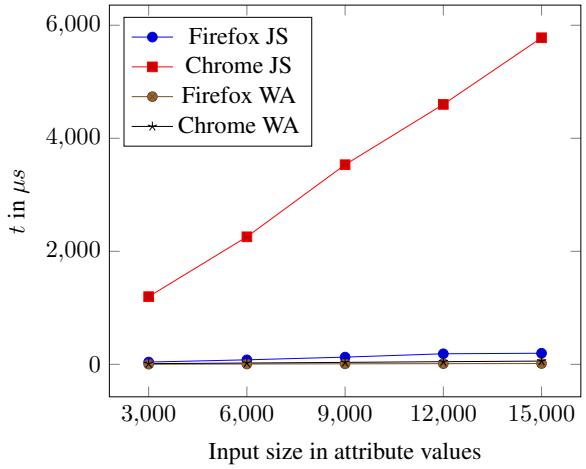


Figure 8: Comparing the execution times at the union of two relations

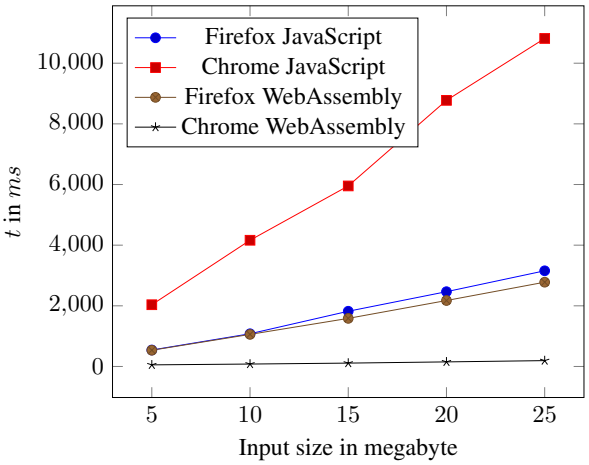


Figure 10: Comparing the execution of TPC-H 6 benchmark

of 11 for Firefox and even a factor of 95 for Chrome in comparison to their WebAssembly equivalents. We also see differences between the two WebAssembly variants, but these are not as big as they are in Javscript.

The linear growth depending on the size was to be expected as there is just one relation that only has to be read once. The differences in the WebAssembly implementations suggest that they are caused by having different good implementations of memory management inside the browsers as the iterative fibonacci variant also uses the same *for* loops and *if/else* structures but causing much smaller differences in the execution times. Same holds for the JavaScript implementations but with much greater effects: Firefox is significantly faster than Chrome because the *Int32Array* class was implemented in different ways.

We expect that the union operator will have a linear runtime as both relations will be concatenated according to the column based storing. Hence, they only have to be read once and appended in the result: As a result the output of two relations with m and n rows has $m + n$ rows. In our experiments (see Figure 8), the execution times of the union operator has indeed a linear growth, but this time Firefox is much faster for WebAssembly and JavaScript in comparison to Chrome. Even though Chrome achieves a significantly slower execution time of the WebAssembly code resulting in a difference of factor 100 while Firefox reaches a factor of 14.

The measured execution times of cartesian products follow a reasonable polynomial growth (see Figure 9). However, we also observe much lower runtimes for both WebAssembly functions compared to the JavaScript

ones. We also see that they hardly differ between the two browsers: While the WebAssembly variant has speedup of factor 10 in Mozilla Firefox, the difference in Google Chrome is about factor 100. Even though Firefox was faster than chrome in all cases.

The polynomial execution time is caused by the cartesian product: In our experiments, both input relations have the same size and contain m rows and n columns. This means that every relation contains $m \cdot m$ attribute values. When applying the cartesian product to them this results in m^2 rows and $2n$ columns with a sum of $2 \cdot m^2 \cdot n$ attribute values.

All execution times of the TPC-H 6 query (see Figure 10) rise linear as the input data size grows but their gradient differs. Hence the required times of the JavaScript variant running in the Chrome browser rises from 2036 milliseconds at the 5 Megabyte data set to 20816 milliseconds at the 25 Megabyte data set by a factor of 5.31 while the data set grows by factor five. Similar results are achieved in Mozilla Firefox with a factor of 5.81 and an increase from 543 to 3155 milliseconds on the same data sets. Just a little bit faster is the WebAssembly engine in Firefox with 535 and 2779 milliseconds execution times, resulting in a growth of 5.19. Using Chrome's WebAssembly implementation there are big differences finishing the processing after only 53 respectively 193 milliseconds while the execution times also grow less with a factor of 3.64.

As this query uses one relation that is being iterated only once, it's running time growth is linear. Despite to the fact this query uses the same language features we already evaluated before, we see differences to the evaluation of a single operator: One is that the execution times between JavaScript and WebAssembly are closer to each other, another one is that the Chrome browser is the fastest. The first difference is related to the usage of the *DataView* class for the JavaScript implementation: It seems to have a better performing implementation than Chrome, by offering a performance for the linear memory similar to WebAssembly's. The second difference can't be explained this way as the memory access stays the same.

However, we have do admit that the compiler needs between 4 and 18 milliseconds depending on the browser to translate and instantiate the program code into WebAssembly bytecode while the instantiation of a new JavaScript function by using *eval()* is done within a millisecond. This difference needs to be amortized at execution time. It is the case for the TPC-H 6 query when using an input with a size of at least one Megabyte.

Additionally we detected very different times of the browsers when executing JavaScript code, especially when dealing with ArrayBuffers. Hence it can be worth the effort to compare and select from the available options. This is also true for the Big Data Engine that consistently fills the ArrayBuffers with new input and may be a possible bottleneck in the processing pipeline.

5 Summary and Conclusions

We realized a JavaScript-based compiler for a small fragment of the Kotlin language using the parser generator ANTLR4. The developed compiler is running inside the browser and generates WebAssembly bytecode to be directly instantiated in the browser and executed afterwards. The in- and output of data uses the shared memory between WebAssembly and JavaScript by splitting them in single bytes, which are stored in a column oriented format.

To evaluate the performance of the compiler and the data format we implemented two mathematical functions and query 6 of the database benchmark TPC-H in JavaScript as well as in our Kotlin-like language. Afterwards we run benchmarks with various input data sizes using the web browsers Mozilla Firefox and Google Chrome.

In our experiments the compiled WebAssembly bytecode is faster executed compared to equivalent JavaScript-code variants. Indeed the speedup of executing the generated WebAssembly bytecode versus the JavaScript variant grows with increasing size of the input data. With these experiments, we show that using WebAssembly bytecode greatly speeds up query processing in browsers and are a key for a competitive query processing in the envisioned Browser Cloud. By achieving high performance results, our contributions are also a great step towards query and Big data processing in the browser in real-time.

In our future work, we will work on our multiplatform engine to support the most important platforms and include also native desktop and server targets like Linux and windows. Furthermore, we will realize the Browser Cloud. We have also plans to develop a hybrid multiplatform database, where data storage and query processing spans over different platforms at runtime.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. Row-stores: How Different Are They Really?" in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 967–980.

[2] Apache Software Foundation, "Apache Parquet," 2016. [Online]. Available: https://parquet.apache.org/

[3] Appcypher, "Awesome WebAssembly Languages," 2018. [Online]. Available: https://github.com/appcypher/awesome-wasm-langs

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[5] J. Bewersdorff, *Gibt es Klassen in JavaScript?* Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 126–141.

[6] N. Bhatti, J. Goff, W. Hassan, Z. Kovacs, P. Martin, R. McClatchey, H. Stockinger, and I. Willers, "Object Serialisation and Deserialisation Using XML," in *10th International Conference on Management of Data (COMAD 2000)*, 2000.

[7] S. Brodersen, "Multi-Platform Big Data Analytics Engine with Code Generation for the Java Virtual Machine and the Browser," Master's thesis, University of Lübeck, 2018.

[8] Databricks, "Catalyst Optimizer - Databricks," 2018. [Online]. Available: https://databricks.com/glossary/catalyst-optimizer

[9] K. Graffi, D. Stingl, C. Gross, H. Nguyen, A. Kovacevic, and R. Steinmetz, "Towards a P2P Cloud: Reliable Resource Reservations in Unreliable P2P Systems," in *16th International Conference on Parallel and Distributed Systems*, 2010, pp. 27–34.

[10] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, May 2011.

[11] N. Heath, "The 3 Next Big Programming Languages: GitHub's Rising Stars for 2018," https://www.techrepublic.com/article/the-3-next-big-programming-languages-githubs-rising-stars-for-2018/, 17th October 2018, accessed 21st June 2019.

[12] A. Hinchman, "Kotlin & Data Science: A Budding Love Story," https://towardsdatascience.com/kotlin-data-science-a-budding-love-story-ad366a633213, 30th October 2018, accessed 21st June 2019.

[13] G. Horváth, N. Pataki, and M. Balassi, "Code Generation in Serializers and Comparators of Apache Flink," in *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS'17. New York, NY, USA: ACM, 2017.

[14] janino compiler, "Janino - A Super-Small, Super-Fast Java Compiler," 2001. [Online]. Available: https://janino-compiler.github.io/janino/

[15] JetBrains s.r.o., "FAQ - Kotlin Programming Language," 2016. [Online]. Available: https://kotlinlang.org/docs/reference/faq.html

[16] Microsoft, "Sprachintegrierte Abfrage (Language-Integrated Query, LINQ)," 2017. [Online]. Available: https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/

[17] MonetDB B.V., "MonetDB - Column-Store Pioneers," 2004. [Online]. Available: https://www.monetdb.org/

[18] F. Nagel, G. Bierman, and S. D. Viglas, "Code Generation for Efficient Query Processing in Managed Runtimes," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1095–1106, Aug. 2014.

[19] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, Jun. 2011.

[20] O. M. Oliver Vornberger, "O-Notation," Universität Osnabrück, 2000. [Online]. Available: http://www-lehre.informatik.uni-osnabrueck.de/~ainf/2000/skript/node39.html

[21] Oracle Corporation, "MySQL - The World's Most Popular Open Source Database," 1995. [Online]. Available: https://www.mysql.com/

[22] T. Parr, "ANTLR (ANother Tool for Language Recognition)," 1992. [Online]. Available: https://www.antlr.org/

[23] E. Peise, D. Fabregat-Traver, and P. Bientinesi, "High Performance Solutions for Big-data GWAS," *Parallel Comput.*, vol. 42, no. C, pp. 75–87, Feb. 2015.

[24] T. G. N. Peter Mell (NIST), "The NIST Definition of Cloud Computing," https://csrc.nist.gov/publications/detail/sp/800-145/final, 2011, accessed 21st June 2019.

[25] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar, *Peer-to-Peer Cloud Provisioning: Service Discovery and Load-Balancing*. London: Springer London, 2010, pp. 195–217. [Online]. Available: https://doi.org/10.1007/978-1-84996-241-4_12

[26] D. Rickert, "Parallel Programming Models in a Browser Cloud Organized as a P2P Network via WebRTC," Master's thesis, University of Lübeck, 2018.

[27] A. Rossberg (editor), "WebAssembly Core Specification," W3C Proposed Recommendation, https://www.w3.org/TR/wasm-core-1/, 2019.

[28] B. Terlson (editor), "ECMAScript® 2018 Language Specification," http://ecma-international.org/ecma-262/, June 2018, accessed 21st June 2019.

[29] The WebRTC project authors, "WebRTC," https://webrtc.org/, accessed 21st June 2019.

[30] Transaction Processing Performance Council, "Tpc-homepage v5," 1988. [Online]. Available: http://www.tpc.org/default.asp

AUTHOR BIOGRAPHIES

Sven Groppe earned his diploma degree in Computer Science in 2002 and his Doctor degree in 2005 from the University of Paderborn. He earned his habilitation degree in 2011 from the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, an open-source Semantic Web database, and one of the project leaders of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. He is also leading a DFG project on GPU and APU acceleration of main-memory database indexes, and a DFG project about Semantic Internet of Things. He is also the chair of the Semantic Big Data workshop series, which is affiliated with the ACM SIGMOD conference (so far 2016 to 2019), and of the Very Large Internet of Things workshop in conjunction with the VLDB conference (so far 2017 to 2019). His research interests include databases, Semantic Web, query and rule processing and optimization, Cloud Computing, acceleration via GPUs and FPGAs, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Niklas Reimer earned his bachelor's degree from the University of Lübeck in Medical Informatics in 2019. He wrote his bachelor's thesis about compiling queries into WebAssembly bytecode for faster query processing inside of the web browser. His interests include federative networks, databases and health care data standards.

APPENDIX

Listing 2: Naive approach for a multiplication function in JavaScript

```
function NaiveMultiplication(input, k) {
    this.input = input;
    this.k = k;
}
NaiveMultiplication.prototype.multiply = function() {
    this._multiply(this.input);
}
NaiveMultiplication.prototype._multiply = function(input) {
    let index;
    for (index in input)
            input[index] *= this.k;
}
let iterations = 100000;
let numbers = {a:42, b:17, c:23, d:27, e:49};
let mul = new NaiveMultiplication(numbers, 2);
let start = new Date();
for(let i = 0; i < iterations; i++)
    mul.multiply();
let end = new Date();
let time = end.getTime() - start.getTime();
console.log((time / iterations) + " milliseconds average");
```

Listing 3: Generate code for a multiplication function in JavaScript

```
function CompilingMultiplier(input, k) {
    let codeString = "this.multiply = function() {";
    let index;
    for (index in input)
        codeString += "input." + index + " *= " + k + ";";
    codeString += "}";
    eval(codeString);
}

let iterations = 100000;
let compileStart = new Date();
let numbers = {a:42, b:17, c:23, d:27, e:49};
let mul = new CompilingMultiplier(numbers, 2);
let compileEnd = new Date();

let runStart = new Date();
for(let i = 0; i < iterations; i++)
    mul.multiply();
let runEnd = new Date();
console.log(
    (compileEnd.getTime() - compileStart.getTime()) + " milliseconds compilation, "
    + ((runEnd.getTime() - runStart.getTime()) / iterations) + " milliseconds average run time");
```

Listing 4: Generated JavaScript function for multiplication

```
this.multiply = function() {
    this.data.a *= 2;
    this.data.b *= 2;
    this.data.c *= 2;
    this.data.d *= 2;
    this.data.e *= 2;
}
```