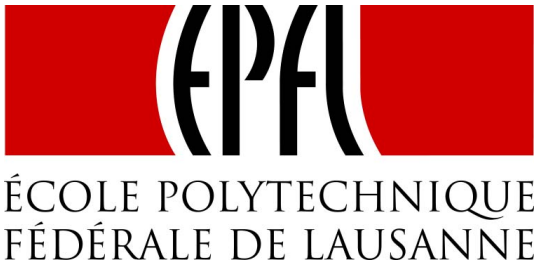

High-performance PL/SQL interpreter for Graal using Truffle

Hugo Kapp

Master Thesis

Department of Computer Science

EPFL

Work done during an internship at

Oracle Labs

Professor Martin Odersky

EPFL Supervisor Denys Shabalin

Industry Supervisor Laurent Daynès

Spring 2017

Abstract

In the age of Internet and ever-growing storage facilities, performance of database operations are critical. Much work has been done in the past decades to improve significantly the execution speed of data queries. To this end, specialized query languages have been designed, which allow for very specific, fine-tuned optimizations. However, such languages tend to be limited in expressiveness, and make the writing of simple data validation or processing tasks very difficult. Procedural extensions to such querying languages lift this constraint, but their additional generality makes them a more complex target for optimizations.

We present in this thesis a more efficient way to execute procedural programs inside the database, using modern *just-in-time* (JIT) compilation, code specialization and aggressive speculative optimizations. Our work is based on the Truffle language framework and the Graal VM, two cutting edge techonologies developped at Oracle Labs. We implement a subset of the PL/SQL language, the procedural extension to SQL in Oracle databases using these tools. We show that the combination of dynamic optimization and code specialization can improve performance over the best PL/SQL execution engines available today. Furthermore, we assess the possibilities and benefits of type specialization inside a statically-typed language, that has a very complex type system. We also demonstrate that it is possible inside runtime-specialized systems to change between different data representations for the same data type at runtime, and that this can be used to improve performance significantly. Finally, we show that, using the Truffle framework and the capabilities of the Graal compiler, it is possible to write such execution engines with a small amount of high-level Java code.

Acknowledgements

I would like to thank Professor Martin Odersky for allowing me to do my Masters thesis in his prestigious lab.

I would then like to thank Denys Shabalin for his supervision during this project, and helping me keep track of time and due dates during my internship.

I am most grateful to Laurent Daynès for helping me throughout this project. His vast knowledge on both databases and compilers were very helpful to aid my understanding of the systems at hand. I wish him all the best for the continuation of his project.

Finally, I want to show all my gratitude to my parents, who helped me get through my studies and every stage of my life.

Contents

List of Figures

1 Introduction

Databases today are a central part of any application. They are used to store information about customers, transactions, identifiers and passwords, and much more. With large and cheap memory storage, and a widespread access to the Internet, these tables typically hold millions of records. It is therefore crucial that getting access to this data and performing simple processing steps is as efficient as possible.

Query languages, like SQL, provide a very efficient way of fetching and modifying data. However, their limited expressiveness restricts the set of processing tasks that can be performed efficiently. To lift this restriction, procedural languages also running inside the database are usually added to allow for more flexibility, and to reduce the amount of communication between the database and its clients. Until recently, performance of these languages were not the main bottleneck. Now that databases start to fit in main-memory, their performance is becoming critical. However, the fact that they are full-fledged programming languages, with much more generality than querying languages, makes optimization much harder. Furthermore, frequent communications between the two kinds of languages incurs, on top of context switching overhead, a mismatch in data representation : space-optimized values provided by the database are typically not efficient when it comes to performing generic-purpse computation.

In this project, we look at the Oracle database and its embedded procedural language called PL/SQL, and try to improve its performance. We do so by using the latest *Just-In-Time* compilation technology, code specialization, and runtime data representation specialization. This is done by implementing an efficient AST interpreter using the Truffle language framework, a rich API to write self-specializing interpreters. PL/SQL programs are then run on top of the Graal VM, which uses a new JIT compiler for Java bytecode that aims at bringing more languages to the JVM. It uses aggressive speculative optimizations to execute programs at high speed, and enables foreign languages implemented using the Truffle framework to benefit directly from these optimizations. Truffle and Graal are two open-source projects, actively developped at Oracle Labs. We tackle the specific problem of having data representations not suited for computation by performing runtime data representation specialization. This allows us to use multiple value representations for the same data type, and choose the best fit for the current program at runtime.

This thesis will answer the following questions :

- Can a language interpreter, based on runtime code specialization, improve execution time of the said language ? And by how much ?
- Is this true, even in the very specific context of data-centric languages like PL/SQL?
- Can we use different specialized data representations for the same type, and change them at runtime ? What kind of impact does this have on the execution of the program ?

We complete this introduction by listing the precise goals of this project and discussing previous work. We then set the background for this project. First, we give a quick overview of the PL/SQL language and its specificities in Section 2. Then, we describe the relevant aspects of the Truffle and Graal ecosystem in Section 3. After that, we give details on our approach for a first interpreter using only integer types in Section 4, and complete that basic system with the `VARCHAR2` type in Section 5. We then assess the performance of this

solution in Section 6. Finally, we give directions for continuation of this project in Section 7, before concluding in Section 8.

1.1 Goals

We have multiple goals for this project. The first is to improve the performance of PL/SQL programs beyond what is currently reachable with the options available in the Oracle database.

The Oracle database already provides high-performance execution modes for PL/SQL programs, namely a native compilation option that compiles the PL/SQL programs down to machine code. It also supports many programmer hints and options, which improve the execution time significantly. We show that current JIT technology, using advanced runtime optimizations, can go faster than the solutions compiled ahead-of-time. We demonstrate how runtime code and data specialization can help improve performance by focusing only on the relevant parts of the code. We confirm that, even in very small and data-intensive programs like the ones typically run with PL/SQL, runtime data representation specialization can improve performance substantially.

We limit ourselves to specific programs, which represent the typical uses of PL/SQL inside the database. Contrary to general-purpose languages, that are used to write computation intensive and generally long programs, PL/SQL functions are usually called inside SQL queries. We will focus on small functions, called as UDFs (User-Defined Functions), to demonstrate the advantage of our solution on a typical workload.

However, we do not provide a full re-implementation of the language. Instead, the system designed here should be an incremental replacement of the classic PL/SQL execution engines. We implement a subset of the language only, which should be representative enough, as well as easily extensible. For example, we do not provide support for any SQL queries inside the PL/SQL code, and focus on procedural programs only. This incremental subset approach allows for a smoother integration of our system into the Oracle database, while keeping the original engines as backup.

Finally, this project also serves as an exploration of the Graal and Truffle framework. We show that implementing a high-performance AST interpreter is possible, and made simple by writing everything in a high-level programming language like Java, using the Truffle API. We demonstrate that complex specialization optimization are easy to write, and can be tailored to our needs to improve performance further. We also investigate the benefits of code specialization in a statically-typed context, a technique usually reserved for dynamically-typed languages.

1.2 Previous work

Although many optimizations have been added to the PL/SQL compiler and execution engine over the years, very few are documented. This is due to the technology being a proprietary product, owned by Oracle. The best publication for such matter is the white paper by Charles Whetherell [7]. The author defines the relaxed assumptions on correctness that were made to allow more room for optimization in the PL/SQL compiler. We would reuse his definition of *freedom* if we were to argue about the correctness of our own interpreter.

Other variants of procedural extension of SQL in other databases share the same kind of compilation and execution options. Transact-SQL [15], which is Microsoft's extension to SQL, provides both an interpreted and a natively compiled mode. Like Oracle databases, the latter improves performance radically, and optimizations are performed for

both execution alternatives. PostgreSQL only provides an AST interpreter for its procedural extension, called PL/pgSQL [16]. However, they started integrating just-in-time compilation into the database for executing SQL queries. Melnik [17] presented his results using a JIT compiler for LLVM to speed up query execution. Our approach goes in a similar direction, but we execute procedural programs, and do it from a higher-level representation.

Much work has been done in implenting runtime-specialized compilers and interpreters for various languages. The first example is the SELF [28] compiler in 1992. The concept has been reused to improve the performance of modern, dynamically-typed languages, using JIT-technology. One such project is the IonMonkey [27] JIT compiler for JavaScript, developped at Mozilla, which makes assumptions about argument values of function calls. Most of these projects focus on type specialization, as that is where there is the most room for optimization in dynamically-typed languages. *Type specialization* performs optimizations based on the assumption that the type of an expression is always the same at execution time. Wang *et al.* [24] use this technique to reduce the allocation overhead of the R interpreter, by performing boxing elimination with the runtime type information. Many JIT compilers have been implemented for dynamically typed languages to reap the most benefits from type specialization. We can cite Psyco [20] for Python, Majic [23] for MatLab, and the work of Gal *et al.* on JavaScript [21, 22]. Our work uses the Truffle framework and runs on the GraalVM, and many of the aforementioned languages have already been implemented using the same combination : Graal.JS [13] for JavaScript, ZipPy [26] for Python, FastR [25] for R ...

This work has been done in the context of the Walnut project [3]. Its goal is to integrate new programming languages into the Oracle database with high execution speed, using the Graal VM and Truffle framework. Their main focus to this day is on the JavaScript language, adding database bindings and interactions on top of the Graal.JS implementation. This is very close to what we present here with PL/SQL. However, there are major differences. First of all, they target a dynamically-typed language, which is not our case. We therefore show that the concept of specialization also has many benefits in a statically-typed context. Furthermore, we can account for the very speficic uses of PL/SQL and its tight integration with SQL and the database, for example in the representations used for various data types. We can then use this information to devise specific optimizations, which is not possible in a generic-purpose language like JavaScript. Finally, our prototype is a proposed replacement for something that already exists in the database. Therefore, it is, and must be, and improvement over the existing system. This is a very different approach than trying to integrate something new into the database, with no reference and no backup to hold on to.

2 The PL/SQL language

We give in this section an overview of the PL/SQL language, and explain its most relevant features for this project.

2.1 History

Today, databases are everywhere. They support the data we fetch from the internet, stores large amounts of data for companies, and keeps track of user and customer informations. But the concept of databases is not a recent one. The term appeared in the 1960's, first as "data-base", and eventually became a single word. But the databases changed a lot since then, and the ones we are accustomed to are very different than the ones used in the past.

The vast majority of databases in use today are what is called *relational databases*, which were invented by Codd in 1970 [1]. This new way of storing data uses multiple different tables which consists of a number of columns. Each column define the kind of elements it stores. Tables and columns are usually referred to by names, and are created using a specific language, called DML (Data Manipulation Language). This language is specific to the database system used. This basic system has been extended over the years, but its foundations remain.

```
CREATE TABLE emp (name VARCHAR2, salary NUMBER);
```

name	salary
'John Johnson'	2000
'Ali Baba'	4000
'John Doe'	NULL
'Sarah Connor'	8000

Figure 1: Example of a database table, and its corresponding DML statement

Information can then be retrieved from such tables using a *querying language*. The most popular and widespread querying language today is SQL, which stands for Structured Query Language. Its most basic feature is the `SELECT` statement, which allows to fetch data from a specific table under some user-defined constraints. This was the tool of choice of database users and managers to get the information stored in their database, and it still is today.

However, as is the case for every querying language, SQL has many limitations. Even though it allows for a wide variety of constraints, that can then be set on multiple tables at once, and lets the query issuer decide on which columns' data to fetch and which to ignore, and even apply basic operators on these result, some more complex queries can just not be done. On another level, code reuse is impossible, as queries are stand-alone programs, and are self-contained. Because SQL is not a programming language, it does not have variables or any kind of control-flow operators. Finally, SQL is very specific and distant from traditional programming language. Database users then need to learn SQL and its specificities before being able to use it fully.

2.2 A database embedded language

As described above, the need for a programming language that could manipulate the data stored and generate queries was strong. When faced with this problem, programmers did a very simple thing : they used their preferred programming language (e.g. C or

```
PROCEDURE raise_all(fixed NUMBER, ratio NUMBER) IS
    raise NUMBER;
BEGIN
    FOR employee IN (SELECT name, salary FROM emp)
    LOOP
        raise := employee.salary * ratio + fixed;
        UPDATE emp
            SET salary = employee.salary + raise
            WHERE name = employee.name;
        send_raise_letter(raise, employee.name);
    END LOOP;
END;
```

Figure 2: Example of a PL/SQL program issuing SQL queries

Java), established a connection with the distant database as the client would have done manually, and start issuing queries as strings over the network. This solution, though very clean because of the separation between the various systems, and the reuse of existing languages and compilers, led to poor performance and network saturation.

To fix this issue, the idea was to add, inside the database, a language that would be specialized for handling data and issuing queries. Because this language runs inside the database, nothing needs to be sent over the network. Everything remains local to one machine (or a cluster of machines), which dramatically increases performance. In Oracle databases, this language is PL/SQL, where "PL" stands for Procedural Language.

PL/SQL is a statically-typed imperative language, with a support for SQL queries and data, table and tuple manipulation. It is based on the Ada programming language [18], and has very similar syntax. It is compiled ahead-of-time into a bytecode language that is then interpreted by a virtual machine, called the PVM (for PL/SQL Virtual Machine). It has all the common features of an imperative language, like conditional branches, loops, variables, function and procedure calls. It is also possible to define packages in PL/SQL. Many more features and constructs have been added in the following versions, like records and even objects. For a complete description of the language, its syntax and semantics, please refer to the Oracle PL/SQL documentation [5].

The tight coupling between the database and the language, and the fact that PL/SQL is tailored for data management and retrieval makes it a good target for very specific optimizations, and many have been added to the compiler over the years [7].

2.3 Specificities & differences with general-purpose languages

As a programming language tightly integrated with the database, PL/SQL has many specificities compared to general-purpose programming languages.

First up, the programs and use cases are typically data-centric. This means that the ratio between code and data is inverted : very small programs may deal with millions of database entries, whereas classical programming languages have huge codebases but handle small amounts of data. In this sense, making optimizations is much more specific, as each program run will be very short, but still has to be very fast. The question then is "how much room for optimization do we have, if the ran programs are only a couple of lines long ?".

Even in this data-centric use of PL/SQL, there exists two main use-cases for the

```
FUNCTION pretty_print(salary NUMBER) RETURN VARCHAR2 IS
BEGIN
    CASE
        WHEN salary >= 1000000
            THEN RETURN TO_CHAR(salary / 1000000) || 'M';
        WHEN salary >= 1000
            THEN RETURN TO_CHAR(salary / 1000) || 'K';
        ELSE RETURN TO_CHAR(salary) || '$';
    END CASE;
END;

SELECT pretty_print(salary) FROM emp;
```

Figure 3: Example of a procedural PL/SQL program, and how it can be used in a query as a UDF. The '||' is the concatenation operator.

language. The first acts as a scripting language for SQL, with simple constructs. These programs vary widely in length, but typically have very few computation code in them, and mostly issue SQL queries. The program shown in Figure 2 is an example of such use case. In this scenario, the performance of the PL/SQL execution is not that important. Most of the time will be spent executing the SQL queries, or switching context between SQL and PL/SQL.

In the second use case, PL/SQL functions act as UDFs (User-Defined Functions). We will call UDF calls to a (usually simple) PL/SQL function inside an SQL query. The benefit here is the ability to write the processing or condition of the SQL query in a simpler way, using typical programming language syntax. This also allows developers to reuse code in different queries. In this case, the SQL and PL/SQL engines must both be fast, and switching context between the two should be as lightweight as possible because the function call will be performed for every entry of the table. This is the case we will consider in this project. An example of this use case is given in Figure 3.

Another major difference is the fact that everything is done in the database, from the compilation to the execution. The user only sends the source code to the database, and waits for it to be compiled and stored there. Because everything is done and stored inside the database, it must follow the same rules as normal database elements. This means that the compiled code stored inside the system could be modified or removed concurrently to its use. This could be triggered by the function being modified, replaced, or totally removed, and can be done by any user who has the appropriate rights. These concurrency issues are taken care of by the database system, which performs the checks and take the appropriate action. But it means that a lot of runtime checks must be added, on top of the same kind of checks for data dependency. This makes the system much more complex and impacts performance.

2.4 Type System

The type system of PL/SQL is an extension of SQL's type system. This means that PL/SQL accepts and understands all the types of SQL, and adds more specific ones as well. This removes the cost of conversion when passing values between the two languages, which improve performance.

The specificity here lies in the very different kind of goals the two languages have

: PL/SQL is based on computation, and should have memory representation suited for basic CPU instructions, whereas SQL is based on storage, and uses data representation optimized for precision and low memory consumption. These goals are hard to reconcile, as compressed data representations are very hard to perform operations on efficiently. PL/SQL must support these complex data representations, which is a major technical difficulty.

We describe in this section some of the basic data types provided by PL/SQL. We limit ourselves to the types that have been implemented in this project. For further reference on SQL and PL/SQL types, see [5, 6].

2.4.1 Numeric types

PL/SQL has support for different numeric data types. The most general of these types is `NUMBER`. This is also a basic SQL type, and can represent numeric values with high-precision on a wide range of values. This representation can have up to 38 **decimal** digit precision, and range from $\pm 10^{-130}$ to $\pm 10^{126}$. Values of type `NUMBER` have a complex in-memory representation, which is optimized for storage size and efficient comparison operators. However, this means that arithmetic operators on such values is very expensive.

To achieve better performance on computation-intensive programs, PL/SQL has introduced other numeric types, more suited to computation on classic processors. These types are not part of the SQL standard, which means that conversion from `NUMBER` to these specific PL/SQL types must be performed. We will only focus on the integer types in this project, but PL/SQL also has more efficient types for floating-point values.

The first integer type added is `PLS_INTEGER`, also called sometimes `BINARY_INTEGER`. As one can induce from the secondary name, this integer type, rather than being stored as a decimal number in memory, use the standard binary 32-bit integer representation, manageable by classic CPU instructions. However, this type still has specific semantics. First, it is a nulleable type, so operations on `PLS_INTEGER` must check for the `NULL` value and use null-semantics if appropriate. These kind of values are also guaranteed not to overflow, so every operation that could potentially lead to such a corner-case must check the result for overflow, and raise and exception if it occurred. These added checks on every operation lead to more expensive operations than what could optimally be achieved.

The second integer type is `SIMPLE_INTEGER`. As `PLS_INTEGER`, values are represented as standard binary 32-bit integers, but has the additional `NOT NULL` constraint. `SIMPLE_INTEGER` is one of the few primitive types that cannot be `NULL`. The arithmetic operations on this type can be done using classic CPU instructions, because no null-check nor overflow-check must be performed. It is therefore the most efficient data type for computation-intensive integer programs, granted that the input values are not `NULL`.

2.4.2 VARCHAR2

To represent string values, SQL developers have access to a variety of types, with their differences and specificities. Some of them must explicitly declare their maximum size, to ensure that the size of the column is known and fixed. Among these types, we chose to focus on the `VARCHAR2` type because it has variable-length values, allowing more room for optimization.

In PL/SQL, the type `VARCHAR2` can be used directly, not requiring any conversion from values coming directly from the database. The language requires local variables of type `VARCHAR2` to declare their maximum length, and pre-allocates the required size for short strings (up to 4000 bytes). However, function parameters of the same type do not have

such constraint, and are only bound by the maximum size of a `VARCHAR2` value (32′767 bytes). This mismatch incurs much room for optimization, as programmers will usually just use the maximum size, or some very big value, when they need to manipulate the parameters and store them in a local variable.

In a PL/SQL program, `VARCHAR2` values are immutable, and passed by value, not by reference. Furthermore, such values can only be modified through a predefined set of operators and builtin functions. This means that the programmer is never given access to, or knowledge about, the internal representation of the string value. This allows for much freedom when implementing such type, which we exploit heavily in our implementation, as discussed later in Section 5.

2.4.3 NULL semantics

To fully support the type system of SQL, PL/SQL must also handle the `NULL` value. This value can appear in the database when a value is unknown. The value of the said column is then `NULL`, which is a special value that can be stored in any column, regardless of the type. To prevent such corner-cases, it is possible to add a `NOT NULL` constraint when declaring the column's type.

In addition to representing the absence of value, or unknown value as it is also referred to in the documentation, `NULL` also has a complete set of semantics. This is different from most programming language, which typically raise an exception when an operation is performed on an unknown value. In PL/SQL, all operators must be able to handle this value without raising an exception, and define a safe return value. In most cases, the operator returns the `NULL` value when one of its operands is `NULL`.

But this value does not only encode the unknown value. For the `VARCHAR2` type, an empty string is also considered null. This is due to long-lasting implementation decisions, and must be accounted for. The semantics for such values are even more specific, as some operators treat it as an empty string, but others treat it as `NULL`. For example, `CONCAT('a','') = 'a'`, but `LENGTH('') = NULL`.

2.5 Compilation process

PL/SQL programs are compiled ahead-of-time. The basic unit of compilation can either be a standalone function or procedure, a package, an anonymous block[1] or more complex constructs. For every compilation, space is allocated in the database system to store information about the compilation unit. Among other things, its name, kind and compiled code are kept.

The PL/SQL compiler is a classic multi-stage compiler that uses multiple different intermediate representations (IR). The first of these is the DIANA, which is a tree-shaped IR. Because PL/SQL itself is based on the Ada programming language, PL/SQL's DIANA is also similar to Ada's DIANA, which stands for Descriptive Intermediate Attributed Notation for Ada [8].

The Diana acts as an annotated AST for the compiled program. It is therefore a complete representation of the input, in a more practical format than the PL/SQL source. Is is augmented with type information on all the nodes, and other analysis results. When it is complete, the Diana tree is stored inside the database.

Because programs are not hardly linked together, compiled programs could become invalid. For every call to a foreign function, the runtime system must check that the code

[1] Anonymous blocks are unnamed portions of code that must be executed directly, as they can't be stored or referenced. They can appear directly in SQL queries, or in the command line.

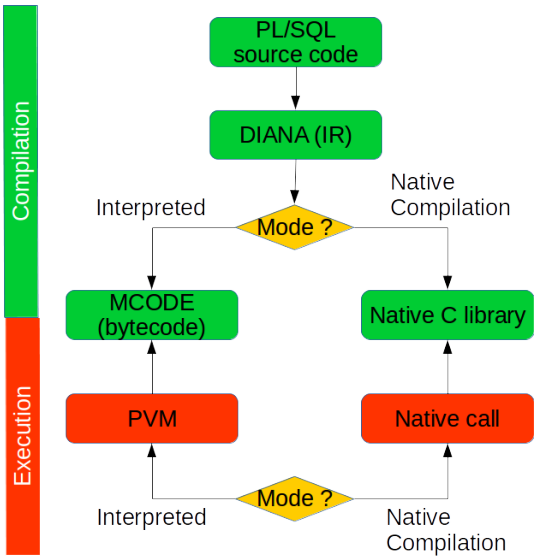

Figure 4: The compilation model of PL/SQL

is still there, that it is the same as the last call used (for data integrity), and that the signature is the same. If any of these conditions does not hold, then execution is stopped, and an exception is raised. This is one of the reasons why the Diana is kept.

2.5.1 Interpreted mode: the MCODE

The first execution mode for PL/SQL is interpreted mode. In this mode, the source code is compiled down to an Oracle bytecode, called MCODE. It is then read and executed by an interpreter, called the PVM for PL/SQL Virtual Machine. Although this bytecode has the advantage of being portable across databases, it has poor performance, mostly due to the interpreter not using dynamic compilation technology.

To improve performance, many classic optimizations passes have been added to this compilation phase, which leads to much better bytecode produced [7]. The interpreter also has very specific bindings to the database for efficiently issuing SQL queries. But this is not enough to reduce the huge interpretative overhead of this execution mode.

2.5.2 Native compilation

To improve performance, it is possible, since Oracle 9.x, to compile the PL/SQL source program down to machine code, using the *native compilation* option. This improves the performance of PL/SQL execution significantly, and can show 2x to 7x improvement over interpreted mode, depending on the input program. This is done by generating C code from the input program, then compiling this to machine code and linking it to the Oracle libraries.

This approach is still limited by some factors. First, to ensure correct semantics of the newly compiled programs, the generated code still has to perform shared library calls with the interpreter. These calls are expensive, and hard to optimize, even by modern compilers, due to the complexity of the whole system. Second, the improvement can only occur on the procedural section of the programs, as SQL statements in the program must still be sent to the SQL engine for execution. Due to the size of database tables, and the very fast speed of execution of native programs, these SQL statements really represent

the bottleneck for such PL/SQL programs. Execution speed improvements are the most significant when the ran program does not contain any SQL queries, and is better for longer running programs.

In this project, we will try to improve performance over native compilation. We will focus on the use cases that already provides the best performance improvement in the native compilation case, for the same reasons. This means varying-length PL/SQL programs, called as UDFs in an SQL query. This ensures that we get significant performance improvements while still running typical PL/SQL workloads.

3 The Graal / Truffle ecosystem

This section gives insight into the environment used for this project, comprised of the Truffle framework and Graal VM.

3.1 The Graal VM

3.1.1 A new JIT compiler for Java bytecode

The GraalVM is an open-source project developped at Oracle Labs [2]. It is a high-performance Java Virtual Machine, using a very aggressive optimizing compiler, based on speculative runtime optimizations.

Graal itself is a new compiler for Java bytecode. It is intended not only for Java programs, but for any programming language that targets this intermediate representation. The Graal VM is a complete Java Virtual Machine, that uses Graal as (one of) the JIT compilers. In GraalVM, the underlying virtual machine is the Hotspot JVM, but it is important to note that the Graal compiler is completely decoupled for its host VM. It complies with the JVMCI API (*Java Virtual Machine Compiler Interface*), but assumes nothing about the virtual machine actually running the code. This allows the compiler to be used in various setups, which will prove useful when we talk about the SubstrateVM.

The compiler is entirely written in Java. This means that, when the JIT is triggered for the original program, the host JVM will interpret Graal's code. After some time, this code will get 'hot' as well, and be compiled dynamically. This compilation can either be done by Graal itself or by another dynamic compiler residing inside this virtual machine.

The major goal of the GraalVM is to provide very high-performance to high-level languages targeting the JVM, using speculative optimizations. To perform these optimizations, it uses its own GraalIR [9, 10]. This intermediate representation is graph-based, in SSA-form, and represents both control-flow constraints and data constraints. Nodes are only bound by their required constraints, meaning that some nodes may not have any control-flow constraint until scheduling occurs. This representation of the program makes the writing of optimizations much simpler than would be on a more classical IR.

3.1.2 Speculative optimizations

The dynamic compiler of the GraalVM makes extensive use of *speculative optimizations*. Through tracing and profiling of the program's execution in the interpreter mode, the compiler is able to decide which branches of the program are very unlikely to be taken when executing in the compiled code.

Using this information, the compiler only produces code for the likely branch. This has two main advantages. First, the size of the compiled code is reduced. Second, this opens the door for many other optimizations. Indeed, the compiled portion of the code is only valid for a specific subset of inputs (the ones that satisfy the condition). This gives a lot of information about the possible types and values handled in that portion of code, which can be subject to virtual call elimination or partial evaluation.

The unlikely branch is not compiled, but instead replaced with a *deoptimization point*. If such a point is reached during the execution of the compiled code, the execution must exit the compiled code, and continue running in the interpreter instead. For this to work, additional information must be stored in the compiled code to reflect the state of the interpreter at the deoptimization points. This information is later retrieved, and the interpreter can safely be started back from this point. The memory overhead of this technique has been looked at in [11].

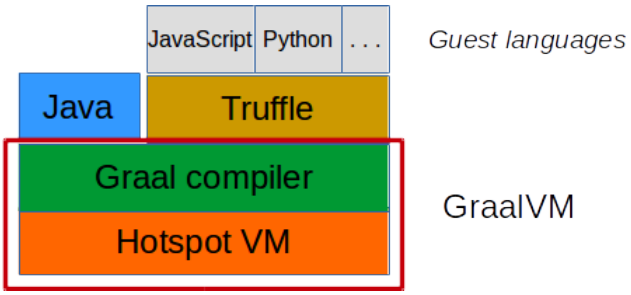

Figure 5: The complete GraalVM + Truffle stack

Reaching a deoptimization point is expensive, because it means switching execution context, and continue running the rest of the code in the interpreter. Therefore, once deoptimization occurred, the compiler will typically re-compile with the branch that caused the deoptimization, to ensure it doesn't happens again.

The condition for deoptimization is called a *guard*. It is handled like normal code in the IR, and thus can me moved in the program by the scheduler. They are typically moved out of loops, and checked as early as possible in the resulting compiled code, in order to minimize the impact of guards on performance.

3.1.3 A multi-lingual platform

Another goal for this project is to provide an efficient multi-lingual (or polyglot) platform. The idea is to share as much as possible the engineering intensive part of a virtual machine implementation across languages, in particular, the dynamic compiler that is crucial to performance. Sharing the same runtime environment and dynamic compiler also make interoperability between languages easier and more efficient.

To solve that open problem, the Graal team decided to bring all the languages to the JVM, while providing a simple way to integrate or re-implement the desired languages : the Truffle Framework. Efforts have also been made in order for the optimizations of Graal to take into account the fact that not all languages executed are necessarily Java at their core, and could use very different patterns.

3.2 The Truffle framework

3.2.1 A language framework on top of Graal

The Truffle framework is anoter open source project from Oracle Labs [12], developped hand-in-hand with the GraalVM project. Its goal is to bring a variety of programming languages onto the JVM, so that they can benefit from the advanced optimizations implemented in Graal. Language implementations should be easy to write, and Truffle provides an API to write AST interpreters for this purpose.

We give in this section an overview of the features Truffle provides, focusing on the ones most important for this project. We will call *guest language* a language that is running on the VM through the Truffle engine. The *guest program* is the input program, written in the guest language.

3.2.2 AST representation

A common attribute to almost every programming language is the concept of expressions and statements. Using this, it is possible to represent every program of a certain language

using an Abstract Syntax Tree (AST). If we want to execute such a program, we can use this AST form. Indeed, every node in the tree represents an operation, and its children supply the arguments to that operation.

Therefore, to execute a program in AST form, each node must do the following :

1. compute the values of the children nodes
2. execute the operation on the subvalues produced by the children
3. return the output value to the parent node

Truffle provides an API to implement such AST interpreters. It also provides a DSL, based on Java annotations, useful for generating boiler-plate code. The representation used guarantees that almost every language can easily be implemented using this platform.

Each node in a Truffle AST must have a method called execute, which returns an Object. Nodes must declare their children using the Child annotation. Nodes also have a single parent, which is found automatically by Truffle when the tree is created.

Nodes can access and modify shared information, like variables states, through a Frame object passed during execution. Guest variables are then allocated on this frame object when the input program is parsed into the AST, meaning that the number and types of all the variables are known before running the program. Each guest variable must declare their storage type, which could be changed at runtime. This type is either a Java primitive type, or the generic Object type. This way, only the necessary space must be allocated, and primitive values need not be boxed.

3.2.3 The concept of specialization

The concept of specialization is key in getting good performances from executing ASTs in the Truffle framework [13, 14]. It is a specific case of node rewriting.

In Truffle trees, nodes can be rewritten at runtime. This means that a specific node in the tree is replaced by another, typically with a more general implementation. In this case, the compiled code must be invalidated, thus requiring these events to be rare and stabilize during the execution of the program. This is useful when one wants to make an optimistic assumption on the input supplied to a node, and use a more efficient method of implementing the operation. If the assumption turns out to be false at runtime, the efficient implementation is replaced by a more general one.

This is exactly what happens in the case of specialization. It allows for an AST node to give multiple implementations for an operation. Instead of replacing the node completely every time, it only "activates" some of the implementation, which still acts as a node rewriting. The choice of which code to execute depends on guards, specified with the specialization, which check the input values provided to the node.

Take the example shown in Figure 6. It shows the implementation in Truffle of a simple "+" operator for a dynamically-typed language. In a simple implementation, the node would need to handle all cases. Instead, we can split the code, and add type-specific variants. Depending on the type of the inputs, Truffle will choose which code to execute at runtime.

Though nodes are simpler to write this way, this is not the only benefit from using specialization. Once a node has been executed, and one of its specialization taken, Truffle will make the assumption that every following execution of this node will take the same path. Only the guard for this specialization is checked, and everything else is replaced by a deoptimization point. This allows the node to be very fast for one specific set of

```
class Add extends Node {
    @Specialization
    int addIntegers(int a, int b) {
        return a + b;
    }

    @Specialization
    double addDoubles(double a, double b) {
        return a + b;
    }

    @Specialization
    String concatStrings(String a, String b) {
        return a.concat(b);
    }
}
```

Figure 6: Example of a simple Truffle node

inputs. If the assumption is invalidated at runtime, i.e. the guard is not satisfied for some input, then re-specialization is triggered. All the guards are checked for this input, and the specialization taken is added to the list of currently active specializations for this node. Only these two specializations are now checked at runtime.

If a node has too many active specializations, Truffle will deoptimize the node. The implementation will revert back to the general case, checking all specializations for applicability on every input. This is the slowest case of all, but the system is based on every node usually having the same kind of input during execution. Notice that like Graal, Truffle uses here a speculative optimization. In fact, this is eventually optimized out by the dynamic compiler. It will only compile the active specializations, and replace the others by a deoptimization point.

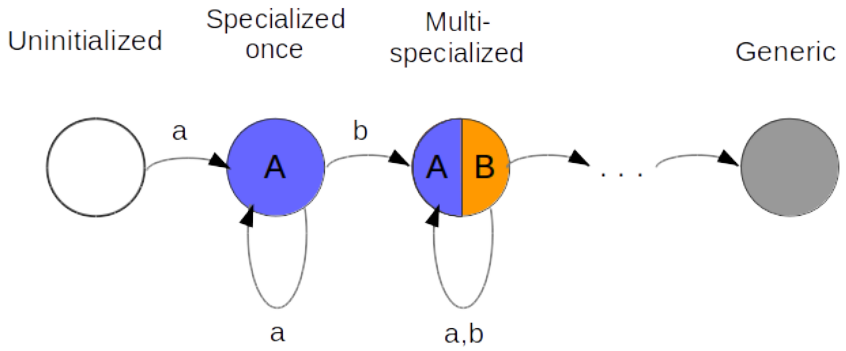

Figure 7: DFS representing the specialization process of a node

The Truffle API allows for many ways to define specialization guards. The simplest one is to use the types of the input. The example used above shows such guards. Truffle automatically reads the signature of the declared specialization, and infers type guards for these. It will then execute the children of this node, and check their return type for a currently active specialization matching it, making type casts if necessary. It is also possible to write guards as a Java expression on the input values. Note that one can combine multiple types of guards to narrow down the specialization to a very specific

case.

3.2.4 Compiler primitive access

As a high-performance framework, Truffle must provide programmers with very specific and fine-tuned tools to take advantage of the underlying compiler. The dynamic compiler in this case being Graal, additional profiling informations are very useful in order to help it make speculative assumptions and optimizations.

First, Truffle allows the specification of deoptimization points directly in the Java code. Any code following this instruction need not be compiled, as it must be executed in the interpreter. This is a simple way to make an explicit speculative assumption. It can be used to wrap very rare cases.

Truffle also provides a way to declare a value as being final while in the compiled code. This is again a speculative assumption, because such values need not be declared as final fields. This constant value can then be used for constant propagation and partial evaluation. On top of that, if the value is an Object, then Graal can inline the virtual calls as it knows the actual type of the value. If such values are to be changed during execution, then it is the programmer's duty to add a deoptimization point first. The compiled code is then invalidated, and will need to be recompiled later with the new value.

Truffle also lets the writer of the interpreter use explicit profiling. Profiles track information at runtime about values, in the form of simple conditions. For example, one could be interested in checking if the value of an input integer is always the same, or if the actual class of an Object value is identical for all runs. On top of profiling values (both the value and its type), it is possible to profile conditions and branches. Truffle provides specific profiling classes which can be used to check if an if branch is taken more than the other, or directly assume that a branch will never occur. These informations are then used by the compiler to make speculative assumption.

3.3 High performance execution

3.3.1 Partial evaluation and inlining

We have already seen, with specialization, that Truffle makes speculative assumptions (well understood by Graal) to improve performance. We present here another use-case, in the partial evaluation of executed ASTs.

Truffle makes the very simple assumption that the shape of the AST is fixed. Although not completely true because it provides support for replacing nodes in the tree on-the-fly, and that nodes rewrite do occur in the case of specialization, we can see that this is a reasonable assumption once the code has stabilized. Once the program is parsed, turned into an AST, and ran a couple of times (to give it time to stabilize), we can expect it to stay the same during the execution. With this very basic assumption, is is possible to reduce interpretive overhead significantly.

In Object-oriented languages, it is known that virtual calls are the source of many performance issues. This obviously happens here, as node execution is triggered by an abstract execute method call. However, because the assumption was made that the AST is constant once loaded, Graal can treat every node of the tree as a constant value. This means that it knows the actual type of all the nodes too. Using this information, it can then inline the execute calls in every parent node, starting from the root, to obtain a single and huge portion of Java code that represents the whole guet program. Optimization on such list of sequential instructions (not containing virtual calls) is rather easy. This aggressive

inlining alone, is enough to remove the overhead of virtual dispatch when executing the AST.

This kind of optimization can be seen as a specific case of partial evaluation, in which the actual type of the receiver object of a virtual call is the constant value. That is why this very aggressive inlining process occurrs during the partial evaluation phase of the Graal compiler. In the general case, one would like to avoid inlining all of these functions, as that creates massive, monolithical fragments with duplicate code. However, this optimization is crucial to the performance of Truffle. For this reason, it is triggered only when compiling Truffle ASTs (other programs still benefit from regular inlining).

3.3.2 Boxing elimination

Another major performance issue with this kind of interpreters is the overhead of boxing primitives values.

Because the return type of a node cannot be statically determined, values produced are passed as generic Objects. When passing primitive values, this means allocating a boxed object for it, then unboxing it at the use location to perform the operation on the value itself. The result is then wrapped, yet again, into an Object box, before being returned to the caller. This adds a lot of runtime overhead. On top of that, if a lot of such boxes are allocated, then repeated execution of the garbage collector can cause massive slowdowns as well.

To solve this issue, primitive values should not be boxed at all. The first way Truffle can avoid these is very simple : add static type information in the Java interpreter's code. To do that, one can add type-specific execute methods to their hierarchy of nodes. When relevant, the caller can then apply the correct method, and be ensured of the return type, removing the need for a box to be allocated. An example of such a use case is shown in Figure 8. If the called node cannot supply a value of the expected type, then it must throw a specific caught exception, which contains the generic value as an Object. The caller is then responsible for handling this exception and its value.

Another way to eliminate boxing is by using an automated way. Graal implements that, by performing an escape analysis phase. The goal of this analysis is to figure out if some value can escape its local context, and be used as a generic object somewhere. If a primitive value is found to escape, then the box must be allocated, otherwise some foreign access could be invalid. If the value is proved to stay local only, then the primitive value can be accessed directly in all use sites, and no box is created. Though this optimization can be hard to carry out in the classic AST form, it is much simpler when performed on the completely inlined version of the same tree, as discussed in 3.3.1. In this case, the scope for the escape analysis is the local code block, e.g. loop body or condition branch, or the Truffle program itself (the whole tree, starting at the root).

Using the exact same analysis and concept, it is possible for Graal to avoid the allocation of other objects, not only boxes around primitives types. If a class is instantiated, and the newly created value is found not to escape the local context, then allocation can be avoided. The fields of the object are replaced with local variables, and access to the fields are made directly on these values. In this case, the object is said to be *virtualized*.

With this, allocation for most of the local values can be eliminated. In fact, most values remain local to the program only, and are never visible outside. Notable exceptions are values used as arguments for function calls in the guest language, return values of guest language functions, and values stored in another objects' field.

```
abstract class Node {
    abstract Object execute();
    abstract int executeInt()
        throws UnexpectedResultException;
    abstract boolean executeBoolean()
        throws UnexpectedResultException;
}
```

(a) Abstract node defining the typed execute variants

```
class UnboxedIf extends Node {
    @Child Node condition;
    @Child Node thenBranch;
    @Child Node elseBranch;

    Object execute() {
        try {
            boolean conditionResult = condition.executeBoolean();
            if (conditionResult) {
                return thenBranch.execute();
            }
            else {
                return elseBranch.execute();
            }
        }
        catch (UnexpectedResultException e) {
            // handle type error
        }
    }
}
```

(b) 'If' node using typed execute

Figure 8: Example of typed execute usage

3.3.3 Example

We present here a small example of an interpreter's run using Graal and Truffle, to explicit the concepts described. We use a language with a single node, presented in Figure 6, which performs a "+" operator for a dynamically typed language. It performs an addition on numeric inputs, and concatenation for strings. We also consider the very simple program "$a + b + c + d$", whose corresponding AST is shown in Figure 9. We will show that even with simple interpreter code written in a high-level language, it is possible to get close to the optimal machine code we would expect.

For the sake of this example, say the values a, b, c and d are always integer values. In the first run, the nodes 2 and 3 specialize for the integer type, and so does node 1. This means that Truffle now makes the assumption that the inputs will always be of type integer, and only check for this condition.

After a certain number of run, the JIT compiler will trigger. Reusing the assumptions made by Truffle, it will make speculative optimizations on the inputs being of type integer. Only the integer addition code for the nodes needs to be compiled. A guard is added to check that the input are indeed of type integer, and a deoptimization point set in the invalidating case.

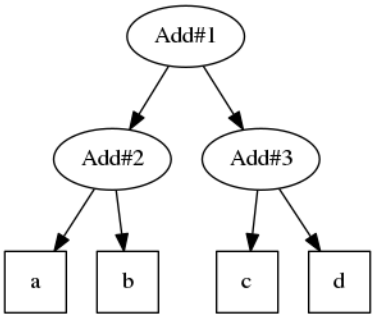

Figure 9: AST for a simple input program

As we said earlier, Truffle also makes the additional assumption that the tree does not change. Once again, using this assumption, the compiler is able to inline the code of the whole tree, starting at the root node. It can now remove the guards in the code for node 1, because the only possible output of nodes 2 and 3 are integer values. Boxes can also be eliminated after escape analysis, and replaced with uses of int values.

What remains after these optimizations is a code that does the following :

1. load the values for a, b, c and d
2. check that they are of type Integer
   - if not, deoptimize
3. unbox the values
4. perform addition on primitive integers

We can see here that the code is not completely as we could expect. Some guards still remain, but need to be for correctness. Because we have no knowledge about the way the values are loaded, they are loaded generically, and need to be unboxed. Note that this could be optimized away, depending on the nodes used to load these values. But the cost of boxing / unboxing is removed for the root node, and arithmetic add operations are performed sequentially without any additional checks.

Finally, let us consider the case where, after the code has been compiled in the way described above, some guards become invalid. For this example, we will assume the values suddenly become String values. In this case, the guards of nodes 2 and 3 are invalidated. This triggers deoptimization, and Truffle must re-specialize the said nodes. From now on, guards for Integer and String must be checked. The same then occurs for node 1.

Once this new tree has been run many times, a new compilation will occur. But the room for optimization is now smaller, as less is known about the values in the program (they could either be integer values or strings). The code will be less performant, and need to take the decision on whether to add or concatenate at runtime. However, this case is assumed to be somewhat infrequent, even in untyped languages, because a single node in the AST will usually treat the same types.

3.4 SubstrateVM

3.4.1 An embeddable VM

Although the GraalVM does provide very good performance to Java and foreign languages programs, it does have some limitations. Java Virtual Machines in general are known

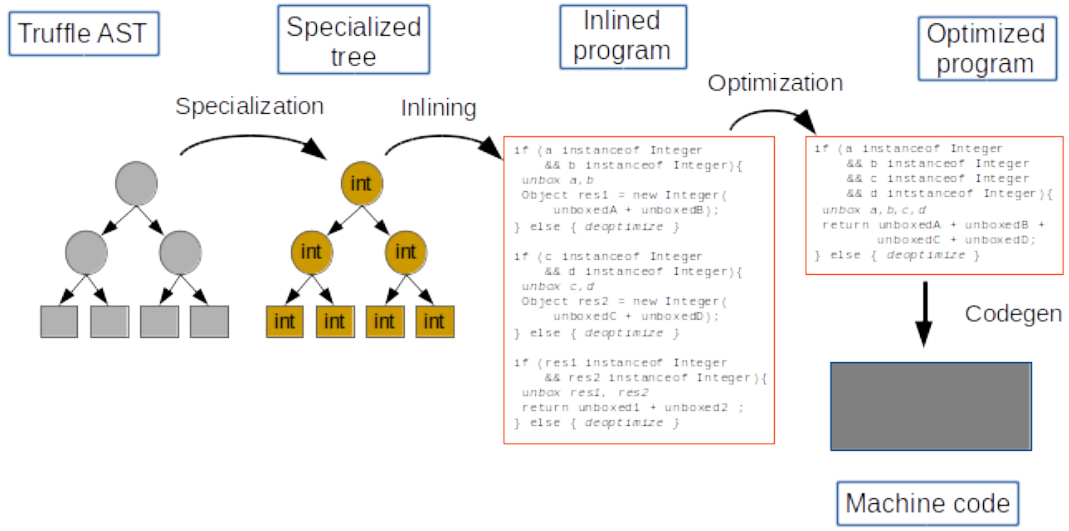

Figure 10: The steps taken by the interpreter code before it becomes highly-optimized machine code

to have high memory consumptions, and long warmup times. These problems are not acceptable when the goal is to integrate Graal and Truffle inside another system.

To be able to integrate the Graal compiler and the languages implemented with Truffle that run on top of it, the Graal team implemented a new way to execute them : the SubstrateVM (abbreviated SVM). The Substrate VM is a very thin virtual machine that provides the necessary services to run Truffle interpreters. It provides a garbage collector, thread support, and works with most of the JDK library. Its main goal is embeddability, so it should have a very low warmup time and a small memory footprint.

To be embeddable, the SubstrateVM should be self-contained, which means that runtime class-loading is not possible. Further, the SVM is not a JVM, and does not behave like one. Thus, some low-level calls or virtual machine specific knwoledge cannot be used here, like reflection for example. These restrictions form a subset of the Java language, feasible for SVM integration. Sources to be deployed on SVM, as well as any reachable code, must be written in this subset and comply to the restrictions.

3.4.2 A pre-compiled VM

The GraalVM, due to its complex and highly-layered structure, has very long warmup times. When executing a guest language program, multiple artifacts must be compiled before the executed program reaches peak performance :

1. the underlying JVM must get to warmup, and start compiling Graal's code
2. Graal itself must warm up, and start compiling the interpreter code
3. the nodes of the AST interpreter must stabilize for the inputs seen, and stop any deoptimization : this is the optimal code

To solve this issue, the SVM is built as an already compiled entity. This way, the only compilations that take place at runtime are guest language program compilations.

The Substrate VM is also implemented in Java. To build a pre-compiled image, multiple steps are required. First, the relevant sections of code must be found. This include

SVM and Graal's code, on top of the desired Truffle interpreters, and any library these may use. This is done by performing static reachability analysis, starting from these predefined components. The analysis is performed by Graal, which has dedicated modules for static analysis. Once these classes and methods are found, they will be the only ones available in the image. This is due to the requirement of SVM being thin and slef-contained, and removes the burden of checking for and loading missing classes at runtime.

After that, the code is compiled ahead-of-time using Graal. As we said earlier, Graal is not bound to be a JIT compiler in a specific VM, and can be used in many contexts. This is one of them, where it is used as an ahead-of-time Java bytecode to machine code compiler. The resulting image is rather small, as it contains only the necessary parts of the code, with a size between 200 and 250 Mb for the PL/SQL images we built.

The guest program, because it is represented as an AST in memory and loaded at runtime, can only be compiled dynamically. As we showed earlier, this is desirable to benefit from speculative assumptions. This means that runtime compilation can still occur in the SubstrateVM, but it is limited to Truffle programs. To be able to do this, the image must carry specific information about the code that is susceptible to just-in-time compilation. This includes the code of the interpreter's nodes themselves, but also the parts of the Truffle API used by the latter, and generally any code reachable from the AST nodes. It is possible to restrict this set, by 'hiding' some code behind specific Truffle boundaries, provided by the DSL. This can avoid loading massive portions of code from external libraries, which may not benefit a lot from dynamic compilation. Because the complete environment is statically-compiled, warmup times for the guest program are typically very short. The compilation time overhead can be reduced by performing compilation on a separate thread.

3.4.3 Direct C access

Easy and efficient manipulation of native C types and data structures is key to embedding languages in complex host systems. The Java virtual machine provides an interface that can be used to interact with external code and values, namely JNI (for *Java Native Interface*). This solution is hard to use in practice, as writing such code is cumbersome and has poor performances. The SubstrateVM offers specific abstractions to ease the interactions between Java and C code, and does so in a lightweight fashion.

Because SVM is compiled natively to machine code, and is not bound by the restrictions of a JVM, it is possible to make easy and very efficient direct memory accesses. The Substrate VM uses this to provide fast interfacing with external C code and structures. It supports the C primitive types without any conversions in Java, as well as pointer arithmetics. It allows C data structure access and function calls to and from C.

SubstrateVM provides a DSL to declare the C structures used in the Java code, which are then used as Java interfaces. When compiling the image, the Java access methods are replaced with direct memory accesses into the C data structures. This interface-based mechanism allows the Java code to be clear, type safe, and valid for compilation by any Java compiler.

The DSL also provides mechanisms for function calls, to and from C. Java methods to be called from C should be static, and never called inside the Java codebase. Some annotations must then be added to its declaration to make it accessible. The function can then be called in C using function pointers. Java code can also call C functions easily, by providing the name of the target entity. These calls are resolved when linking the image.

This is very useful when interacting with lower-level APIs, and allows fast and efficient interoperability. We use these functionalities in the project to get direct access to the

values provided by the database, without requiring a copy into the Java heap.

4 Implementing a subset of PL/SQL using Truffle

We describe here our approach for implementing a subset of PL/SQL using Truffle. It uses all the tools provided by the framework to achieve high-performance. It also exploits the static typing of PL/SQL to implement improvements based on data representation specialization.

4.1 Naive approach

We first present a simple approach to building an interpreter for PL/SQL, based on how dynamic languages are typically implement within the Truffle framework.

In this system, basic operators (e.g. "+", "−", "<") are represented by a single node which can handle every input. It declares specializations for the valid pairs of type that the operation can be performed on, and defines the semantics of the operation in this case. An example of such a node is given in Figure 11.

For this system to work, every type in the guest language (PL/SQL in this case) must have a corresponding, and unique, Java type to represent it. Values of the guest language can only be represented by these types. We therefore have to find a mapping between PL/SQL types and Java types. A natural way to do so is to create a class for every type, and box all required information inside.

We would like to use Java primitives types when possible for such mappings. `SIMPLE_INTEGER` naturally maps to `int`, but `PLS_INTEGER` cannot map to the same Java type. Being a nulleable type, values of the type `PLS_INTEGER` need additional information to represent the possibility of a value being `NULL`. A simple solution is to create a class for this type, comprised of a 32-bit integer value and a flag for `NULL`.

However, this solution incurs a lot of overhead on boxing and unboxing, and increases memory consumption. For operations on non-null values, the integer value is unboxed, the operation is performed, and the result is the boxed again in a new object. For null values, the flag must be checked, but no object needs to be allocated : the null value found can be reused. We expect values to be non-null most of the time, which has the highest cost. Note that Graal should be able to virtualize most of these object allocations. This would make us very reliant on Graal correctly performing its escape analysis, and suffer a heavy cost when some value appears to escape.

4.2 `PLS_INTEGER` and `NULL` handling

To alleviate this problem, we would like to be able to use primitive `int` values to represent non-null `PLS_INTEGER` values, and a specific `NULL` value otherwise. However, this cannot work in the system described. Because the operators choose semantics using the input Java type, they cannot differentiate between `PLS_INTEGER` and `SIMPLE_INTEGER`, which have different semantics on overflow (the first raises an exception while the other does nothing). This is the typical implementation of dynamic languages in Truffle, where the input type defines the semantics used.

We use the fact that PL/SQL is a statically-typed language to fix these issues. Using type information from PL/SQL, we build a statically-typed AST. Every node can only handle values of their fixed type. We can then safely map different PL/SQL types to the same Java type. In this system, we can also use different Java representations for the same PL/SQL type. This will be the basis for data representation specialization.

Both non-null values for `PLS_INTEGER` and `SIMPLE_INTEGER` use the primitive integer type. Further, we add a specific type to represent the `NULL` value, implemented as a singleton.

```
class NaiveAdd extends Node {
    @Specialization
    int addSimpleIntegers(int a, int b) {
        return a + b;
    }

    @Specialization
    PlsInteger addPlsIntegers(PlsInteger a, PlsInteger b) {
        if (a.isNull()) {
            return a;
        }
        else if (b.isNull()) {
            return b;
        }
        else {
            int safeAdd = Math.addExact(a.value, b.value);
            return new PlsInteger(safeAdd);
        }
    }
}
```

Figure 11: Naive add implementation for PL/SQL

Specializations in the operation nodes can now be performed on primitives types when possible, reducing the stress for escape analysis and boxing elimination.

This system is aimed at helping Graal produce more optimized code. First, removing explicit boxing lightens the dependency we can have on Graal effectively escape-analyzing all the values, and virtualizing the corresponding objects. On top of that, Graal understands more easily the primitive types and their respective boxes than hand-crafted re-boxing. Note also that type information is easier to propagate for the compiler than flag values in an object's field. This helps partial evaluation and constant folding, especially concerning type guards in Truffle specializations.

Finally, this solution replaces typed null values with an untyped generic `NULL` value. This makes the writing of nodes and conditions simpler, and requires less checks. However, this also requires writing one node per type and per operation. Although this can seem tedious, the Truffle DSL helps greatly in reducing the amount of work required, and reduces the number of possible specializations for each node. This duplication is only limited to semantics-based operators, some other nodes need not care about the type (e.g. write, read).

4.3 Storage model

To represent program state, PL/SQL uses typed variables. Program variables in Truffle are stored in a specific object called the Frame, and passed along during execution of the nodes. Values must be stored in pre-allocated slots in the frame. These FrameSlots must know about the type of value that will be stored : either primitive, e.g. int, boolean, or the generic Object case. They do not store information about the actual type of objects stored, but type profiling could be added when reading from the frame.

The typing of the frame slots poses a major problem for the system we described. Indeed, if a `PLS_INTEGER` value is stored, the type of the frame slot should depend on the actual value : Object slot for the `NULL` value, int slot for the others. Because compilation

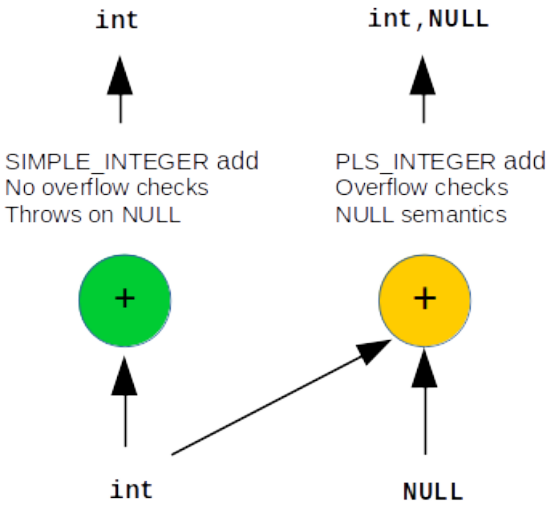

Figure 12: Statically-typed AST nodes for addition

```
class PlsIntegerAdd extends Node {
    @Specialization
    int addNonNull(int a, int b) {
        return a + b;
    }

    @Specialization
    NULL addNullLeft(NULL a, Object b) {
        return a;
    }

    @Specialization
    NULL addNullRight(Object a, NULL b) {
        return b;
    }
}
```

Figure 13: Code of a statically-typed add node for `PLS_INTEGER` values

optimizations depend on the types of the frame slots, changing it causes deoptimization.

One solution would be to optimistically use integer slots until a `NULL` value is found. This event would trigger deoptimization, and switch the slot to a generic Object one, able to store both the singleton null value and integer values. This is called *frame slot specialization*, and is used in the Truffle implementation of Javascript for example. However, this means that non-null values must now be boxed when stored. This boxing is not escape analyzed by Graal (because the object is stored on the frame), and leads to huge overheads in allocation and garbage collection times.

We use a very simple flag based system to solve this. Nulleable variables are reserved two frame slots: one for the value if non-null, the other for a boolean indicating whether the value is `NULL`. This flag is first checked before returning either the null value, or the actual value stored in the other frame slot. This removes boxing, but adds another runtime overhead (added check when reading) and memory overhead (two frame slots per variable).

Though minimal, these overheads can be removed for variables that cannot be null. PL/SQL defines a `NOT NULL` constraint, which ensures that variables cannot be `NULL`. Some types, like `SIMPLE_INTEGER`, are `NOT NULL` by default. For these variables, we can remove the second frame slot and the added check.

4.4 Language support

For this interpreter, we implemented a subset of the PL/SQL language. This subset is enough to express most programs, but does not contain any of the builtin support for SQL. For more details on the operators listed here, please refer to the PL/SQL documentation [5].

We support most of the control flow operators that PL/SQL provides. This includes conditional branching (`IF`, `CASE`), loops (`WHILE`, `FOR`, unconditional `LOOP`), loop-specific control flow (`EXIT`, `CONTINUE`) and return statements (`RETURN`, with and without a value). The only control flow we do not support are direct branching (`GOTO`) and cursor-based operators (`FORALL`).

For defining programs, we support top-level standalone `FUNCTION`s and `PROCEDURE`s. A PL/SQL procedure is simply a function that does not have a return value. They are usually used for stored procedures.

The types we support in our interpreter are the integer types `PLS_INTEGER` and `SIMPLE_INTEGER`, plus the string type `VARCHAR2` (detailed in the next section). As explained above, we provide full support for the `NULL` semantics of PL/SQL. `NOT NULL` variables are also supported, and slightly optimized as explained above. Incomplete supports for `BOOLEAN` and `NUMBER` have been added, but should be looked at as future work.

We implemented the basic operators for every type supported. This includes arithmetic operators on integers (`+`, `-`, `/` and `*`), concatenation on `VARCHAR2` (`||`) and logical operators on booleans (`AND`, `OR` and `NOT`). We also have comparisons ($=$, $!=$,$<$, $>$, $\leq$, $\geq$, `IS NULL`, `IS NOT NULL`), which work on every type.

Finally, to increase the expressiveness of our subset, we added support for builtin function calls, and provide an extensible framework to allow the simple and incremental addition of new builtin nodes. Some numeric builtins are supported (`ABS`), and more `VARCHAR2` builtins (`LENGTH`, `SUBSTR`, `REPLACE`), as this is the only way to handle and modify `VARCHAR2` values.

We believe this is a subset of enough importance to allow writing meaningful programs. We will test this hypothesis when building micro-benchmarks in section 6.

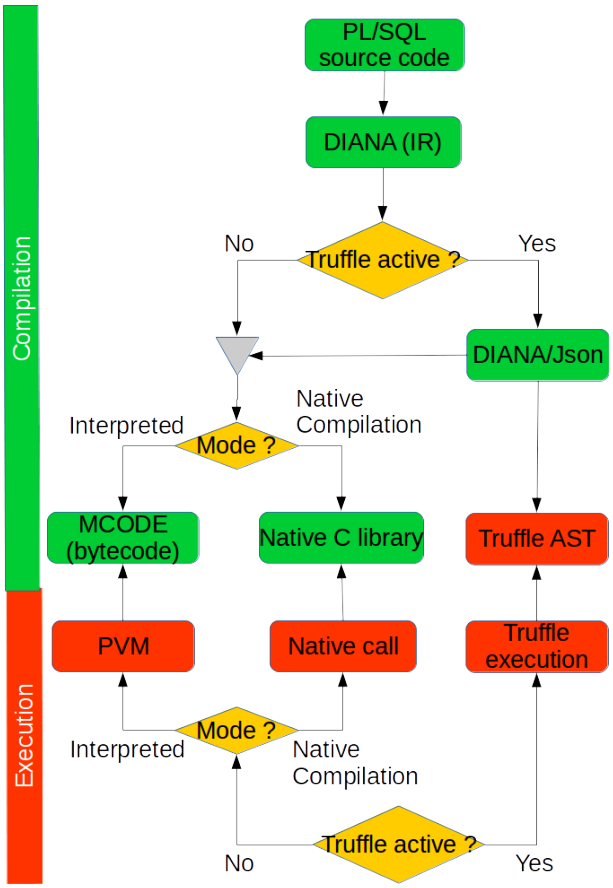

Figure 14: The new compilation model for PL/SQL, with the addition of our Truffle interpreter

4.5 Compilation model

We present here briefly how we compile PL/SQL programs to Truffle ASTs before executing them.

The PL/SQL compiler already provides a way to generate an AST from the source programs. As explained in section 2.5, the first pass in the compiler is to build a DIANA representation of the program. This DIANA form is effectively a typed AST for the compiled program.

To build the Truffle AST, we let the normal compilation process start, and intercept the DIANA tree once it is complete and annotated with type information. We then store it in JSon format to make the parsing easier. This extraction is only performed when the user activated our replacement interpreter, which is done through a system parameter. Note that we aim for non-disruptive incremental changes to the basic PL/SQL execution environment. Therefore, once we extracted the AST, we let the normal compilation continue, as a fallback if the user wants to execute with the classic engine.

JSon programs are parsed on the first execution of the program. From the AST, we generate one Truffle node for almost every DIANA node. We also gather information about the types and variables used. Whenever an operator node is found, we analyze the type of the expression to use the correct statically-typed Truffle node, as discussed when explaining the design.

This solution for compilation is simple and reuses already existing tools.

5 Efficiently implementing VARCHAR2

We describe in this section the design decisions we took for implementing the `VARCHAR2` type. We extend on the concept of data representation specialization to improve performance.

5.1 Design constraints

The `VARCHAR2` type of PL/SQL has many specificities.

First, there is no direct access possible to the underlying representation of such values. PL/SQL only provides a concatenation operator (`||`) and a very complete set of builtins. This is the only way to handle, modify and introspect values of type `VARCHAR2`. Although many languages also provide such an API to work with strings, the internal structure and representation of the values are usually known. This allows programmer to optimize their use of the data type by ordering their operations in the most efficient ways. Languages also typically provide a way to get direct access to the underlying buffer.

PL/SQL does not grant any of these information. Implementation details for this type are largely unknown. No direct access to the underlying structure is provided, mainly due to the lack of a type representing a single character in the type system. This means that programmers must really trust the API, and use it in the way they think is best.

While this can seem to be a lost optimization opportunity for the programmer, it can also be seen as a good and principled abstraction use. This abstraction allow us to represent the values in any way we see fit, completely transparently to the user. This gives us a lot of freedom and room for optimizing the representation of strings.

Another interesting aspect of the `VARCHAR2` type is its uses. We already talked about the very different kind of workload PL/SQL typically has compared to general-purpose languages. This of course applies to strings as well. In generic programs (non-PL/SQL), string values and variables are passed a lot to other subprograms. They are often checked for length or content, and rarely modified. Concatenations occur, but the result is usually sent to another function or procedure. Some values are also stored for a very long time, or act as keys in hashmaps.

Because PL/SQL programs are rather short, the lifespan of `VARCHAR2` values is short as well. If built inside the program, a value is usually sent to another procedure to get it printed on the screen, stored inside a database table, or raise an error. Otherwise, the task of the function could only be to build the string from the input parameters, and return from the call. In all of these cases, `VARCHAR2` values are short-lived, but there is one use case where they last longer. We will call *string-building* a function that constructs a string by aggregating multiple smaller elements. This is usually done inside a loop. The value constructed this way is typically returned to the caller, but shows a use case of longer-lived string values.

While these use cases are very different, we have more precise knowledge at the granularity of operations. After running analysis on more than 3500 typical PL/SQL programs found in already-existing database files, data shows that the concatenation operator is the most used, by far. It represents 76% of all `VARCHAR2` operation uses. If we seek to provide a high-performance implementation of this type, it is clear that we must support a very efficient concatenation operator.

5.2 Classic string implementations

The simplest implementation for a string type is to use a buffer in memory to store the sequence of characters. This buffer should either end with a special terminating character (e.g. null-terminated in C), or be extended with a length. If values are immutable, like they are in PL/SQL, a concatenation operator would typically use a new buffer, and copy the two values. This ensures that the original values are not modified, but is very expensive. In general, copying memory is an operation we want to avoid. Most operations using this representation would typically require a copy, which is inefficient.

Another standard implementation for immutable strings is called *ropes* [19]. This design uses concatenation trees to represent values. Concatenation is performed by creating a new node, and setting the children in the correct order. This provides very fast concatenation operator, and has proved very efficient for certain workloads.

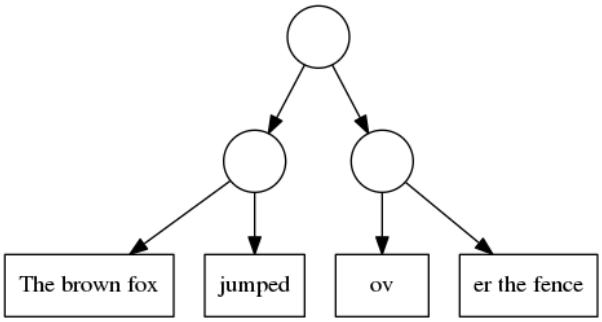

Figure 15: Example of a string using the rope representation

Ropes are better used when handling long-lived and big strings. On shorter values, they tend to have a high-memory overhead. Tree traversal is also an expensive operation. Therefore, ropes might not be well-suited to our use case in PL/SQL, as `VARCHAR2` values are typically short and short-lived. Since the database stores results from call to PL/SQL in it's own flat buffer, rope values cannot be returned without a conversion into a flat representation. This is an expensive operation to perform on trees, requiring a tree traversal that induces many virtual calls that are hard to optimize. Finally, a solution based on trees would require allocation for every value, because the pointed values are abstract classes, which cannot be easily virtualized by Graal.

Because we deem both solutions to be ill-suited for our specific use, we devise a new representation for `VARCHAR2`, tailored for PL/SQL, which uses linked-list for concatenation.

5.3 Native values support and cohabitation

We start with the naive design, and improve on it gradually. Values in this system are represented by a `byte` array for the buffer, and a length attribute. Note that we do no support multiple character sets, so a `byte` implementation is enough for now.

To improve performance, the first thing to do is to avoid copies. The best candidate for that is the copy at the boundary between SQL and PL/SQL. When the Truffle interpreter is called, the call comes from the database environment. The values passed as arguments are therefore C structures and values. This is incompatible with the Java environment, and requires a copy / conversion to take place to represent the same value in the Java heap. Note that this happens twice per function call : once for the arguments, and once for the return values, which must be converted the other way around. This incurs a major cost, especially on repeated calls handling short strings. To avoid this overhead, we propose to

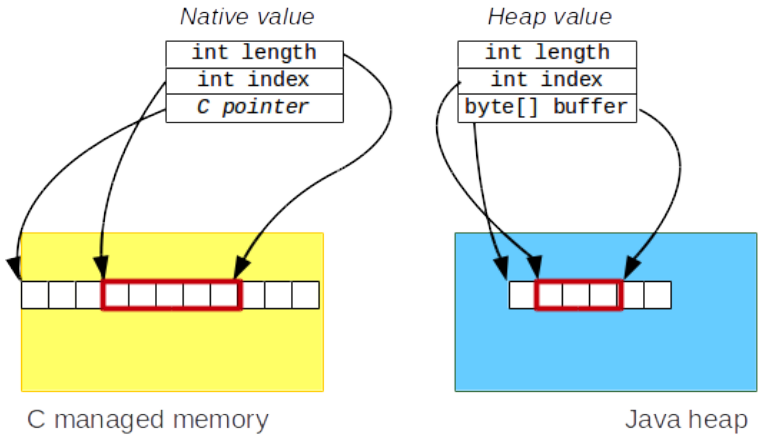

Figure 16: The memory representation for the two types of base values.

support the C values directly. However, we must also support some form of Java heap value, as we can't efficiently create C values. We call *native* values the ones coming from C, and *heap* values the ones lying in the Java environment.

To cope with native C values in our Java code, we use the C API that SVM provides. This allows simple access to these structures in the Java implementation. However, this duality means that values of the type `VARCHAR2` are not bound to a single representation anymore. We already saw a similar example when dealing with `NULL` and `PLS_INTEGER`, so we already have the typed node structure in place, and only need to add more specializations to cover the additional cases. In order to speed up the substring operator, which does not require any copy, we add a start index field to our `VARCHAR2` values. Together with the length, this provides a copy-free substring operation, which only changes the indexes. This will also be useful when implementing other operators.

To recap, we now have two kinds of values in the interpreter to represent `VARCHAR2` : heap values, whose buffer is a `byte` array in the Java heap, and native values, whose buffer is a C pointer. Both are augmented with a length and a starting index to provide efficient substring operations. This solution still requires copy for the concatenation operator, which we want to optimize.

5.4 A linked-list representation for concatenation

We now show how to optimize concatenation using a linked list design. The elements of the lists are *base values*. We call *base value* the representations based on a buffer, like the native and heap values discussed above.

In this design, concatenation is very simple. Each base value is wrapped inside a linked list node. When concatenating two base values, we wrap the two inside a node, and make the left value point the the right one. When concatenating two list-based values, we only need to make the last element of the left list point to the right one. In this case, the right element can also be a base value we just wrapped in a node. Examples of these cases are shown in Figures 17 and 18.

However, this simple design does not meet the immutability constraint. Appending a new element to an already existing list changes the original value by extending it. To fix this issue, we present *horizon-based linked lists*. In addition to the underlying linked list, this representation knows which nodes belong to it. Instead of assuming that the next node in the list always belong to this value, horizon-based linked lists (HBL) keep

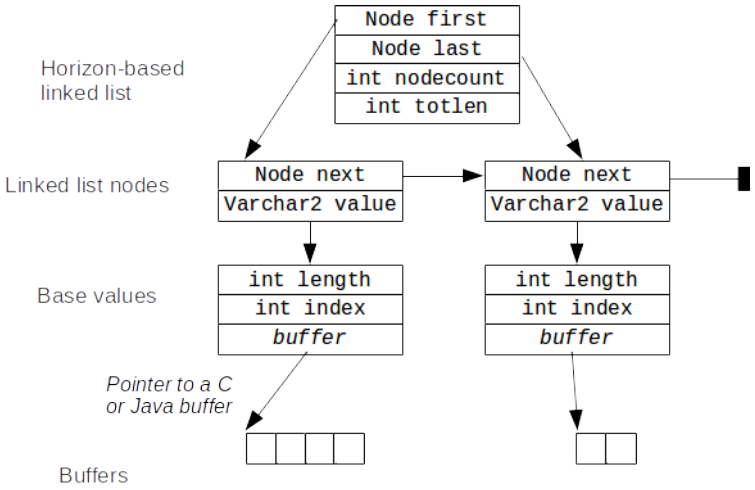

Figure 17: Concatenation of two base values in the linked-list design. The final value is the horizon-based linked list.

track of the number of nodes it contains. This allows for concatenation to remain cheap, but ensure immutability : the original values do not see "past" the number of nodes it contains, so it is safe to append new nodes to it.

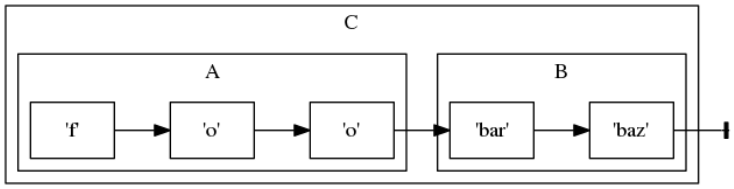

Figure 18: Concatenation of two horizon-based linked lists.
The boxes denote the scope, or horizon, of each value. Notice A and B are unchanged, although their list have been modified.

This design is enough in our case to cover most uses, because string insertions are not possible in PL/SQL. Therefore, only allowing appending and prepending values is enough. To accelerate typical accesses, we keep the total length of the `VARCHAR2` value in a field.

This design is very similar to the rope representation, but has many benefits in our particular case. First, the structure is flat, i.e. always has one layer, which simplifies iteration over the value. This is a common operation which is required when converting the value back into a buffer before returning. No virtual calls are necessary, and we can read the whole value inside a very simple loop. This flat structures also simplifies the escape analysis, and allows the compiler to virtualize the list in simple cases.

5.5 Canonicalization

The design presented so far still has a flaw. While it is possible to prepend to a list indefinitely, only one append is possible. In the case a second append operation is required to the same prefix, the latter must be copied, which incurs a high cost. We now present an optimization for this case, which can be used for other purposes.

First, note that the case mentioned is uncommon. Indeed, programs tend to extend the same string all the time, leading to various forms of a "string building" workload. In this

```
prefix := root || dir || subdir;
patha := prefix || filea;
pathb := prefix || fileb;
```

(a) Appending multiple times to the same prefix

```
fullstr := a || b || c;
x := SUBSTR(fullstr, 1, 5);
y := SUBSTR(fullstr, 6, 10);
z := SUBSTR(fullstr, 11, 15);
```

(b) Multiple substrings on the same concatenation result

Figure 19: Example of patterns that require multiple copies of a list value.

scenario, strings are constructed by appending a lot of small elements to the same variable. However, reusing that same prefix for another concatenation is rather uncommon. This is especially true for the short-lived strings we work on.

But this is hardly the only case which requires our system to perform a copy. Substrings on list values must also perform a copy. Recall that the list values we use are immutable. It is therefore not possible to change the indexes in a base value without changing the value of the whole list. Adding substring indiexes is not a solution either, because it does not allow for the simple concatenation method we used so far (think about appending a substring-ed list to another, the indexes would conflict). Therefore, building a substring of such value requires a copy of the entire list, or at least of the complete `VARCHAR2` value.

This becomes problematic if a program follows the pattern shown in 19b. When multiple substring operations are performed on the same concatenation result, multiple copies of the same value are done. This pattern is more common than the previous one, as using substrings in a loop is the only way to iterate over the characters of a `VARCHAR2` value.

To solve this problem, we first make a few remarks. Note that these problems are due to the same concatenation result being used in different operations. This can only happen if the concatenation result has been stored and re-read later. In our system, this means the problematic value is necessarily coming from a variable read.

The solution is to use a canonicalized form of the value. Every `VARCHAR2` value in our system can be represented as a base value. This is trivially true for base values, but is also the case for lists as we can flatten them into a heap value. Substring and concatenation on these base values is very fast and always works the same way. To remove the duplicate copies, we add a specific node, when building the AST, that ensures the input to substring nodes is always canonicalized. If the said value is produced by a variable read, the flattened value is written back to the frame. Consecutive substring operations are performed on the already-flattened value, which can be read from the frame with no additional cost. We do a similar optimization for concatenation in the conflicting case.

This is still not perfect. It has some room for further optimizations, and creates other inefficient patterns, but it is a step in the right direction, and makes inefficient patterns even more rare. On top of that, with the assumption that we usually handle short strings, flattening lists could be more efficient than copying the list itself.

5.6 Null support and empty strings

A major specificity of the `VARCHAR2` type is the handling of empty strings. In PL/SQL, an empty string is `NULL`. This can be confusing, especially when statements like `'IS NULL ''` do not yield false as expected, but true.

Without diving into the details, let us say that this semantics is very simple to get in the database PL/SQL interpreter, due to the value representation. However, we now have multiple kind of values to represent strings in our Truffle interpreter, and each of them should be checked for emptiness whenever performing an operation. This also means that null checks are way more complex : they are not bound to the singleton value we used so far, but can be represented in a variety of ways.

To simplify this, we disallow the creation of empty strings, and replace these by the singleton null value we used so far. Note that only the creation of heap values must be checked this way. Indeed, empty native strings are directly inputted to the program as `NULL` values. List values must have at least one element, which is a base value, and we already know that base values cannot be empty, so list values cannot be empty either.

This might make the type analysis harder for Graal, as the return type of some operators is only bound by the supertype Object (`VARCHAR2` and `NULL` values have no common ancestor in the type hierarchy). This can become a problem and degrade performance. Note that guard elimination would have become harder anyway, would we have kept empty strings. Some type guards would have been replaced by field-checking constraints, verifying that the length is greater than 0, and such guards are way harder to optimize for Graal than type guards.

5.7 Builtin support

Because builtins are the only way to use `VARCHAR2` values, it is very important that we provide support for a good number of core builtin functions. We currently have concatenation (`||`), `SUBSTR`, `REPLACE` and `LENGTH`. These have been chosen because they are among the most commonly used operators, and because they are enough to program a lot of basic functionalities one would want for `VARCHAR2`.

We already talked about the implementations of concatenation and substring. For the `REPLACE` builtin, we wanted to reuse these same concepts. A replace algorithm, based on a no-copy system was implemented, which would build concatenation lists, alternating between a substring from the input and a pointer to the replacement string. An example of how such an implementation works is shown in Figure 20. However, following the benchmarking results, this implementation had to be changed for a simpler one, copying the result into a buffer. The original algorithm performed poorly, mainly because replacement operations already require the full string to be read, character by character. Therefore, copying each byte is not a lot more work.

In addition to the builtins we implemented so far, we also built an extensible framework, which allows for adding new builtins easily. Conversions from the input types to the ones matching the signature of the builtin are made automatically. Finding and creating a node for such operations, using only the name of the builtin is also simple. Finally, using canonicalization can be a very easy way to implement builtins without providing support for all the different representations we can have in the interpreter. Basic support for the base values is the only thing necessary. Of course, supporting list values without requiring them to be flattened is more efficient, but this solution can be used to quickly expand the set of supported builtins with a small amount of work.

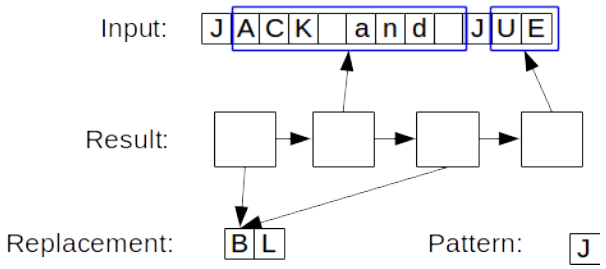

Figure 20: Result of the operation `REPLACE('JACK and JUE', 'J', 'BL')`, using list representation

5.8 Shortcomings

The solution we presented in this section is more performant than a naive buffer-and-copy design. But it does suffer from drawbacks, some of which we already addressed with specific optimizations earlier.

The major drawback of this method is the memory overhead. Indeed, when concatenating, new allocations are made both for the new linked list node and the encapsulating horizon-based list. Each linked list node contains a pointer to its value, plus a pointer to the next node. The HBL contains a pointer to the first node, a pointer to the last node, the number of nodes, and the length of the total string value. The base values contain a pointer to the buffer (be it in the heap or coming from the external C environment) plus a starting index and a length. With 32-bit ints and 64-bits pointers, the overhead for the result of n concatenations is :

$$\begin{aligned} overhead(n) &\leq sizeof(\mathsf{HBL}) + n \times (sizeof(\mathsf{LinkedListNode}) + sizeof(\mathsf{BaseValue})) \\ &= (8+8+4+4) + n\,[(8+8) + (8+4+4)] \\ &= 24 + 32n \text{ bytes} \end{aligned}$$

This is a high cost to pay, but allows us to have very efficient operators implementation. Note that this formula is only an upper bound for the overhead, because measuring the actual memory consumption requires runtime information. This result is based on the assumptions that the sub-results of the concatenation are not shared, and that no base value in that list is shared anywhere else (or appears twice in the list). This last assumption is hard to verify of course, but does not seem reasonable. Making the inverse assumption leads to an overhead of $24 + 16n$ bytes, which is almost half for long lists. Finally, note that the size of the buffers is not taken into account because it does not create overhead (it is necessary for very naive approaches as well).

Even though this method has a high memory overhead, it is possible in certain circumstances that the size of our structure is smaller than the actual string length. This could occur if the string is constructed by always appending the same value. In this case, the base value boxed and buffer is reused everywhere. If the string appended is longer than the size of the linked list node, then memory is saved. As computed, the appended string would need to be longer than 16 characters long.

Another problem with the current design is the fact that there still is some polymorphism in the values. Indeed, linked list nodes could either point to native or heap values. Although length and starting index fields can be factored out and granted access to without knowing the actual type, accessing the value itself (the buffer) requires a type check at runtime. We added type profiling in every code portion that needs to loop through

the lists' elements, and make the assumption that the list only contains a single type of values. In the cases where this is not enough (mixed lists of native and heap values), this problem removes a lot of optimization opportunities for Graal, and makes virtualization much harder. Lack of virtualization in this case can be really damaging for performance, as it would require a lot of allocations at runtime, and incur great stress for the garbage collector. We describe some ways to alleviate this problem further in section 7.

6 Performance evaluation

In this section, we show the performance of our system when compared to the database execution engines. We then validate the different design decisions taken. We close by looking at the memory consumption of the system.

6.1 Database integration

In order to perform a fair comparison of the database and our Truffle interpreter's performance, we must run the same programs, and in the same context. Because normal PL/SQL can't be run outside of the database, we must find a way to integrate our interpreter into the database. This will allow to run typical PL/SQL workloads, as part of SQL queries on big tables, and lead to a working prototype for this project.

We want to be able to perform SQL queries with PL/SQL code in it. The only thing we will be concerned about is making UDF calls from SQL to PL/SQL work. We already discussed some form of database integration when talking about the extraction of the DIANA code in 4.5. We will use the same system parameters to guide the interception of SQL calls to PL/SQL.

First, we must store the JSon representation of the DIANA tree into the database. Using the same code as when we extracted the DIANA to an external file, we adapt it to store the resulting JSon in the database space pre-allocated for the function metadata.

For the rest of the integration, which is not necessarily specific to PL/SQL, but to the problem of executing UDF calls using Truffle in general, we reuse most of the dbjs infrastructure. dbjs is a project at Oracle Labs aiming at integrating and running JavaScript programs inside the database. It provides support for some native constructs and interfacing with SQL, and allows developers to work with the database directly in their JavaScript code, and call JavaScript functions as UDFs in their SQL queries. The programs are run using the Graal compiler, which is very close to what we are trying to achieve.

To intercept the PL/SQL UDF calls in SQL queries, we follow the dbjs approach that allows UDF calls to JavaScript functions to be re-directed to the Graal.JS interpreter. When the SQL query is compiled, we identify all the PL/SQL calls it makes. For each of those, we verify if we already produced JSon for it. In this case, we replace the call by a Truffle stub generated on the fly. The stub also reuses code from the dbjs infrastructure, and is identical to JavaScript stubs until function resolution is needed. This stub is responsible for understanding the database's function call convention, and performing the required conversions before executing the Truffle AST for this function with the correct arguments.

Although we do support native database values, we still need to perform conversions for values of type `NUMBER`. Our Truffle interpreter does not provide support for the latter, but has to accept them. For JavaScript, the stub already contains a set of Truffle nodes that perform the specialization of `NUMBER` values to integer when possible. This implementation is heavily based on code specialization, and is very efficient. Reusing these nodes is useful for checking that the numbers passed are indeed integer values, and then converting the database value into a Java integer we can handle. Once our AST interpreter returns the final result, the stub is also responsible for converting it into a value understood by the database. This might mean copying internal string value representation into a C buffer.

To be able to run our PL/SQL engine inside the database, we build a SubstrateVM image containing our interpreter, and link it to the database. SVM is made for this kind of applications, and manages the calls from the database code to the Java portion.

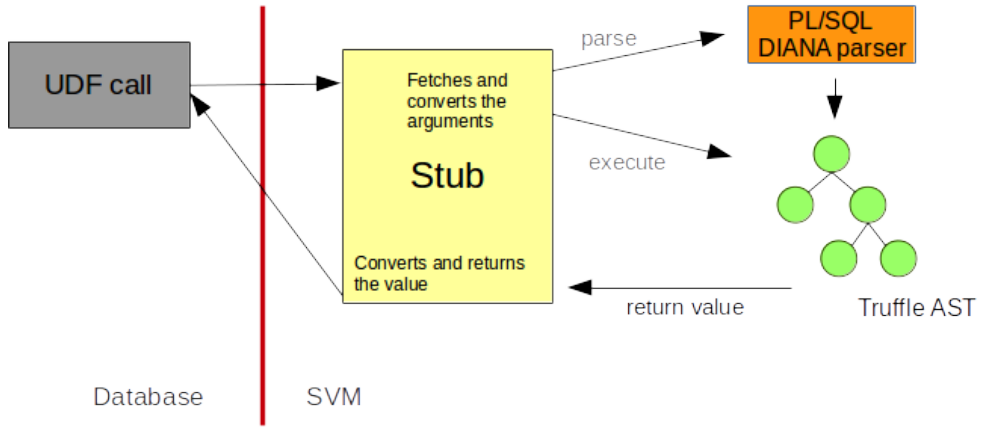

Figure 21: The runtime infrastructure for PL/SQL UDF calls. The function is parsed only on the first call.

6.2 Methodology

We now measure the performance of our system using different metrics. We will showcase the improvement it provides when compared to the current database options, and use these results to validate our design decisions. For the latter, we will specifically measure the cost of null handling, as well as the performance of simpler string implementations.

To get significant results that mimics the typical use of PL/SQL, we try to devise benchmarks that follow the basic workloads of this language. To do this, we evaluate the runnning time of SQL queries that perform UDF calls. These calls are made to PL/SQL functions, and are of varying length. We will benchmark very short programs, as well as longer running ones, using loops. Finally, to simulate the effects of a computation-intensive stored procedure, we also measure the performances of a few standalone programs. We measure performance only using micro-benchmarks, written for this project.

We run these programs as part of SQL queries, either in the `SELECT` part or in the `WHERE` clause. Site of use does not impact performance at all. We run these queries on randomly-generated tables of 10 million entries. We measure the running time of the query itself, and average the results over 5 runs.

We compare the performance of the Truffle interpreter we implemented for this project (noted as truffle) with the different systems that already exist in the database. The two traditional modes of execution are measured, i.e. the PL/SQL interpreter (pvm for PL/SQL Virtual Machine), and the native compilation option (ncomp). To get the best performance out of these, and also to measure the ratio between PL/SQL execution and database-to-PL/SQL context switching time, we use the `PRAGMA UDF` option that PL/SQL provides. This annotation can be added in the code, and tells the compiler that this function will primarily be used in the context of an SQL query. Special care is then taken by SQL when calling this function, to get rid of some of the context switching cost. For this optimization to be possible, some restrictions on the programs are necessary, and are explained further in the documentation. Programs that use this option are natively compiled, and noted pragma.

6.3 Numeric benchmarking

6.3.1 Short UDF case

We first show the performance of the different systems on very short UDF numeric benchmarks. We show in this section the execution speed of `SIMPLE_INTEGER` programs. We recall that this type is the simplest numeric type that PL/SQL provides. It cannot be `NULL`, is represented by a 32-bit integer in memory, and has no specific overflow semantics. This is the most CPU-friendly type, and should therefore be the most efficient.

However, the only numeric type the database understands and supports is `NUMBER`. Therefore, to be able to run programs using a different type, we must first perform a conversion operation. Every program in the set of benchmarks used will have such a conversion as its first operation. Such a conversion will also be necessary when returning the value. We show an example in Figure 22.

```
FUNCTION mul(a NUMBER, b NUMBER) RETURN NUMBER IS
    c SIMPLE_INTEGER := a;
    d SIMPLE_INTEGER := b;
BEGIN
    RETURN a * b;
END;
```

Figure 22: Code for the mul benchmark

The benchmarks run for this experiment are very short, one liners essentially. The first three use only a single operator, the following four show more complex mathematical expressions. Then, const performs an empty call that only returns a constant, to show the cost of context switching, and search implements a very simple compare-and-return function that showcases the performance of comparisons and control flow operators. Execution results are shown in Figure 23. Notice that we do not benchmark the division operator, which would require `NUMBER` support.

We can clearly see that the major costs for the basic systems is given by the context switching, because using the `PRAGMA UDF` annotation improves performance by almost 3.5x for every benchmark. The truffle interpreter is even better, and divides the execution time of the pragma systems by nearly two (1.84x on average).

The performance gain comes mostly from the stub implementation. Indeed, this layer of interaction between the database and the Truffle PL/SQL interpreter performs the conversion from the complex representation of `NUMBER`s to Java-based ones, does the checks for input `NULL` values, and is also written as a Truffle AST. It therefore benefits from specialization, and can make assumptions, for example that the input value is never null, because none was ever found. We also reused existing blocks that implement an efficient conversion from `NUMBER` to integers, using specialization. Finally, the input programs are so short that very few time is actually spent executing the code, most of it is dedicated to conversions and interfacing with the database.

These numbers are very good, but do not tell the whole story. They measure the efficiency of a highly-specialized module we did not write, and gives very few insight into the performance of the different PL/SQL execution engines. To solve this issue, we now focus on more computation-intensive benchmarks, with longer running times, to show the speed of the PL/SQL execution, while still in a believable workload.

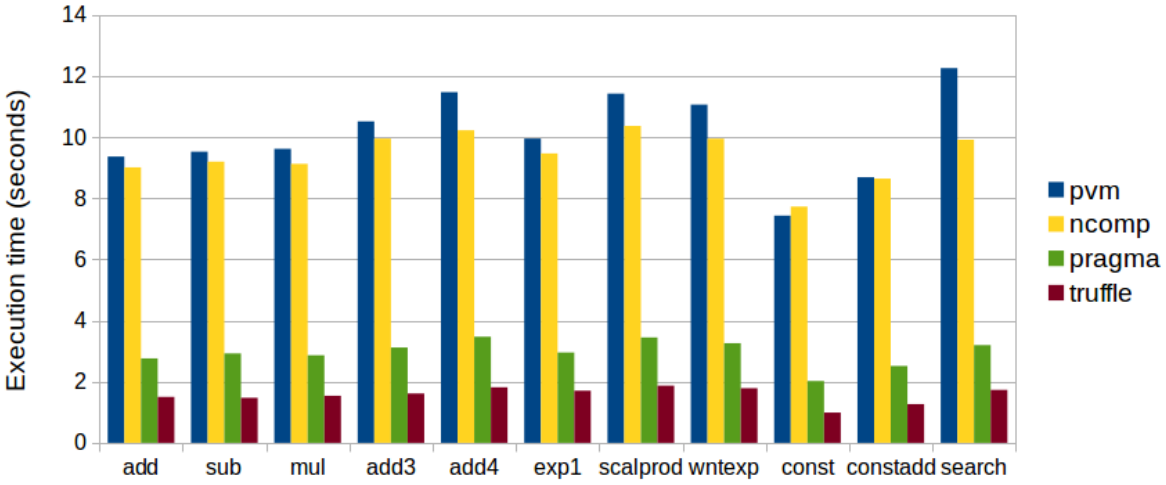

Figure 23: Execution speed of the short `SIMPLE_INTEGER` UDF benchmarks (lower is better)

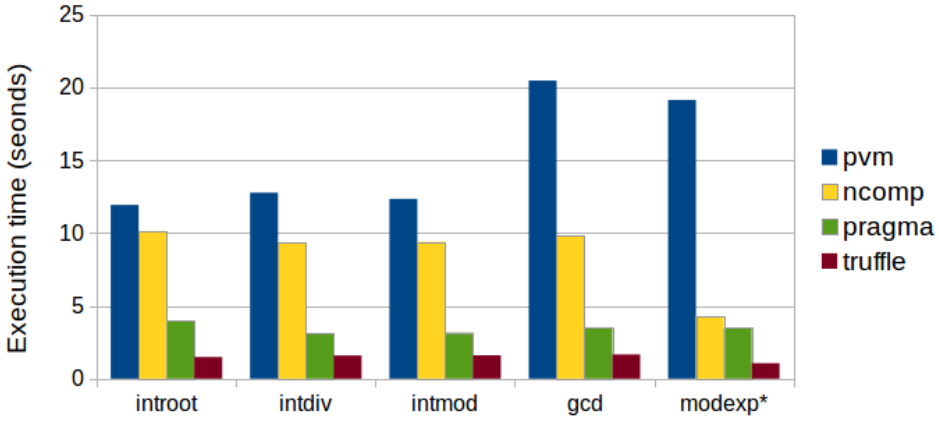

Figure 24: Execution speed of the stored loop-based UDFs
**scaled down*

6.3.2 Loop-based UDFs

Longer running UDFs are very common. Based on loops and temporary, they could be used to filter some results according to a complex criteria. An example could be the following query : `SELECT * FROM datapoints WHERE gcd(a,b) = 1`.

We devise a second set of UDF benchmarks, comprised mostly of loop-based functions, with a varying degree of computation involved. Once again, these only use the type `SIMPLE_INTEGER`. Results are shown in Figure 24. The benchmarks are sorted from the least computation-intensive to the most computation-intensive.

We can already see that the time spent executing the PL/SQL code is longer, because the use of `PRAGMA UDF` onlt gives a speedup of 2.5 for the first benchmark, and does not give any improvement for the heavy modexp benchmark. This observation is reinforced by the improvement given by the ncomp system. Though not providig any speedup in the short UDF case, it is now showing an improvement between 1.20x and 4.50x over the pvm. The speedup provided by our Truffle interpreter also increases. Between 6x and 7x for the previous set of benchmarks, it is now between 8x and 18x for the most computation intensive functions (when compared to pvm). This is a good sign that the PL/SQL

(a) Execution speed (seconds)

	countprimes	countfactors
pvm	37.14	37.50
ncomp	5.63	4.93
ncomp pragma	5.59	4.92
truffle	1.56	1.76

(b) Speedup achieved

		pvm	ncomp
count primes	ncomp	6.6	-
	truffle	23.8	3.6
count factors	ncomp	7.0	-
	truffle	19.6	2.8

Figure 25: Results of the computation-intensive `SIMPLE_INTEGER` benchmarks

implementation we provide has better performance than the builtin systems. If we compare the improvement over the pragma variant, which is the most efficient of the classic options, we get a speedup between 1.95x and 3.25x, averaging at 2.40x. These are very good measurements, and showcase a real improvement over the database implementation of PL/SQL.

6.3.3 Computation intensive benchmarks

To complete this numeric experimentation, we try to get rid completely of the conversion and context switching cost. To do that, we implement two very computation-intensive functions in PL/SQL, that will be called only once on a single-entry table. These programs showcase the performance of our interpreter in another typical PL/SQL workload, computation intensive stored procedures. This experiment will also be useful in determining the actual speedup that comes from executing PL/SQL code with our alternative.

The two functions implemented and measured respectively count the number of primes from 1 to 50'000, and count the number of prime factors every number from 1 to 20'000 has. Figure 25 shows these results. We can see that the overhead of context-switching has been removed completely, as using `PRAGMA UDF` does not give any improvement anymore. The native compilation option also provides its maximal speedup, as advertised in the Oracle documentation, of 7 times the performance of the PL/SQL intepreter (pvm).

The Truffle interpreter we implemented goes further, and reaches up to 24 times the performance of the pvm. This is mostly due to Just-In-Time compilation technology, that uses all the runtime information available to produce very efficient code. Branches and input values can be profiled, and complex three-valued logic based conditions can be reduced to the normal boolean case because no `NULL` condition is ever encountered. More generally, no `NULL` checks must ever be performed in any operator, because it is known that this program only handles `SIMPLE_INTEGER`. Though we did not encode this specifically when parsing the program, Graal found out itself by profiling the seen values. The native compilation option provided by the database cannot do such a task, and uses more general comparison and branching operators that must perform the checks, even though they are unnecessary. Finally, the ncomp system does not reach the optimal performance of a handcrafted C program that would do the same task as implemented in PL/SQL, because it does heavy calls in a shared library to perform most of its operations. These calls are not easily optimized by classic C compilers (e.g. gcc), and thus do not give the CPU-instruction execution speed we could expect. The speedup of our Truffle interpreter when compared to the ncomp option is 3.60x for the countprimes benchmark, and 2.80x for the countfactors benchmark. The speedup achieved is still dependent on the function, but is a very impressive result, keeping in mind the simple AST interpreter we wrote in Java.

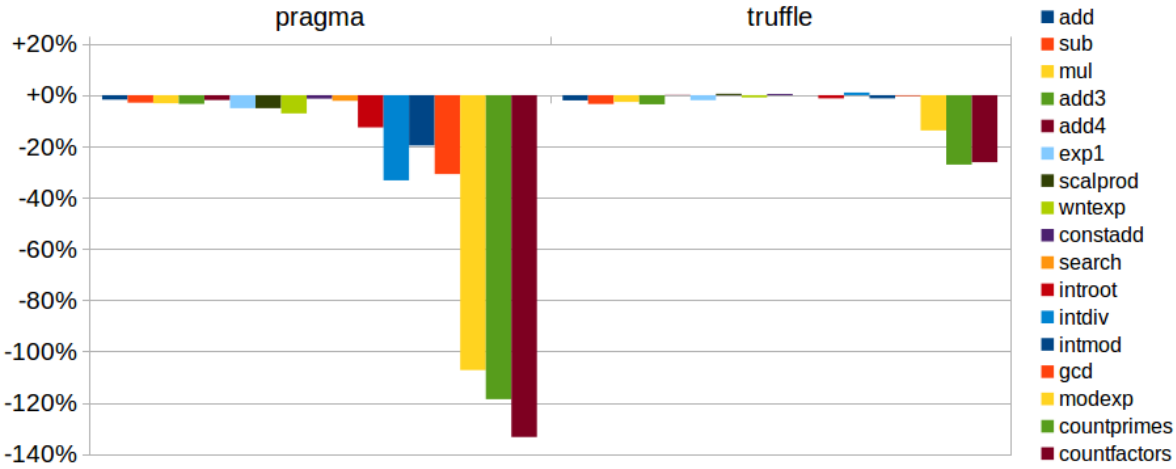

Figure 26: Impact on performance of using `PLS_INTEGER` instead of `SIMPLE_INTEGER`. Execution speeds are compared for the same systems, with the two versions of the programs. Positive values show improvement, and negative values show a degradation of performance.

6.4 Impact of `NULL` handling

6.4.1 Changing the type

To continue our performance evaluation, we now turn to the second integer type we implemented, `PLS_INTEGER`. This will be interesting to see the added cost of null checks and specific semantics in the case of overflows.

The first experiment we run is very simple : we take the exact same benchmarks we ran earlier, and switch the types. Instead of using `SIMPLE_INTEGER`, every input is converted to `PLS_INTEGER` instead. At first glance, that does not seem to change anything, but it does have some impact.

First, because `PLS_INTEGER` accepts the `NULL` value, execution now has to take this possibility into account. This is done through null checks in the basic operators, and should increase the running time. The second reason is the overflow semantics of the `PLS_INTEGER` type. Indeed, values of this type are guaranteed not to overflow. Therefore, every arithmetic operation performed on it must ensure that no overflow occurred. This is slowing down every single operation, and increases the running substantially.

For each benchmark, we measure the execution speed degradation of the pragma and truffle systems, by comparing the running time to the same system on the `SIMPLE_INTEGER` version of the function. This shows the impact on the evaluated systems of changing the type used, without actually encoutering the `NULL` value (queries are run on the same tables, that do not contain any `NULL` value). We measure the degradation, and show the results in Figure 26. All benchmarks previously run are shown sequentially.

The pragma suffers major slowdowns from this change. As we already said, in the short UDF case, very little time is actually spent executing PL/SQL, which explains the small impact on these first programs (less than 10% slowdown). We can see that this impact increases with the amount of computation performed : on the loop-based UDFs, the slowdown goes up to 30%. This is a heavy cost just for changing type which integer type is being used. Finally, the worst slowdowns come from the most computation-intensive benchmarks. The program's execution takes twice as much time to complete when compared to the `SIMPLE_INTEGER` case. This cost is due to the more expensive operations that `PLS_INTEGER` supports. Indeed, the very good performance of this system on the heavy

benchmarks was due to CPU-friendly operations, that could be performed natively. In that case, some of the operators could be transformed directly into a single CPU instruction. However, with the more complex PLS_INTEGER semantics, every operation now needs to check if some of the operands are NULL, and if the operation execution did overflow. This is a heavy cost to add to every operation, which now performs conditional-branching even for the simplest arithmetic expression.

The Truffle interpreter does not have that problem. As we can see, there is some slowdown on almost every program, but it is very small for the simplest programs (less than 3% for the UDFs). For computation-intensive benchmarks, the maximum slowdown observed is 27%. This is still a big price to pay for only changing the data type, but is much smaller than the native compilation, which showed the best performances on these programs. The operators are still more expensive in the Truffle AST, because overflow checks cannot easily be avoided (this would require profiling on every operand of every operator). However, most NULL checks can be removed, except the input ones. Indeed, no such value is ever encoutered, meaning that the only data representation flowing during execution of the program is the primitive int type. This cannot be a null value, and Graal can then make assumptions that this is the only case of interest. This removes all internal checks, which gives a smoother degradation. The initial checks must still be performed on the input values, because no information about these external values coming from the database is known. With this smoother degradation, we reach up to 6x speedup over the best database implementation (pragma), with an average of 2.75x.

6.4.2 Multi-specialized nodes

We now discuss the encounter of the NULL value, and its effects on the Truffle interpreter. This is especially interesting because it showcases the added cost of having multiple specializations active at the same time in some nodes. This requires the node to take the decision on which implementation to use at run-time, and reduces the opportunity for speculative assumptions. Because this is a Truffle-specific problem, we only measure our interpreter for this experiment.

Measuring the impact of the NULL value on performance can be tricky for PL/SQL programs. Indeed, this is a special case that is thought to be quite uncommon, but still has a complete set of semantics attached to it. Comparisons with NULL yield NULL, and branching on NULL takes the false branch. Loops are thus avoided if based on a NULL comparisons, and directly go to the end. This means that for our loop-based programs, the main body of the function is usually not executed at all. Measuring the change in execution speed for various amounts of NULL is then not very useful.

Instead, we will apply the benchmark queries we used so far on similar, but not exactly identical tables. The baseline will be measured on the same tables as before, with no NULL. We then measure the first variant on the same table, with one NULL entry added. This entry has one null field, and all the others remain classic values. The second variant has one added entry per column, with each such entry having exactly one NULL field. An example is shown in Figure 27. Once the programs have been run on these alternative tables, the Truffle nodes have more than one specialization active at the same time. Re-running the program on the same table allows us to measure the time taken by the generic code. This gives us the degradation when the code is not perfectly specialized, and has to take care of all the possible inputs.

The results for this experiment are shown in Figure 28. We can see a small increase in the '1 NULL' case for every benchmark. The 'all-NULL' case is always the slowest case of all. This makes a lot of sense because it means more nodes have actually been affected by

(a) Basic table

a	b	c
1	2	3
-1	0	17
501	7	1042

(b) One-null variant

a	b	c
1	2	3
-1	0	17
501	7	1042
NULL	2	1

(c) All-null variant

a	b	c
1	2	3
-1	0	17
501	7	1042
NULL	2	1
13	NULL	-100
1	503	NULL

Figure 27: Example of the NULL variants of the tables used for the multi-specialization experiment.

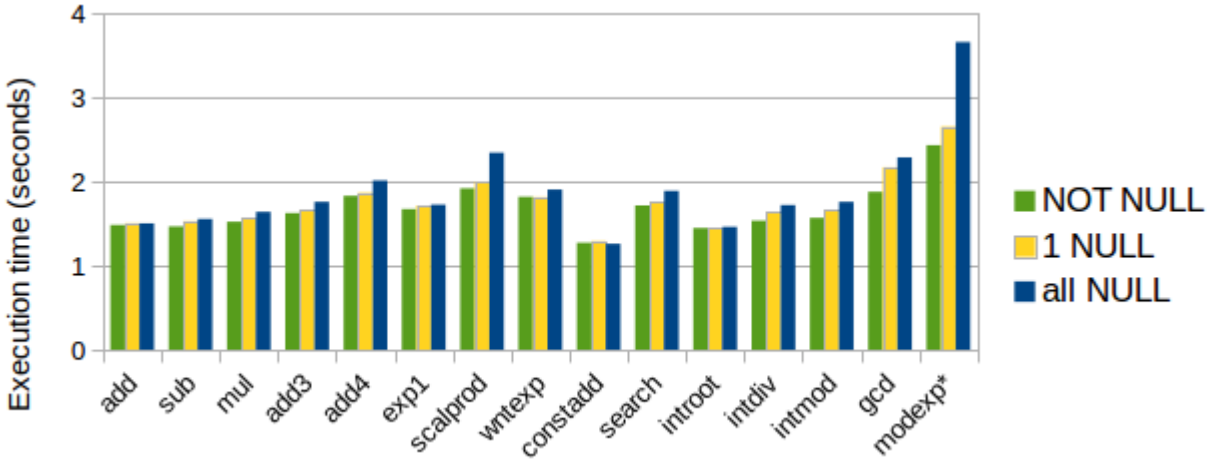

Figure 28: Execution time of the Truffle interpreter in the multi-specialized case. **scaled down*

the NULL values. The middle case is interesting because it shows that a single NULL value, though altering the performances, only leads to a small increase (less than proportional) in execution time. This is due to the fact that only a few nodes have been affected. When all the columns are null one after the other, way more nodes are affected, leading to poorer performances. This change is greater when the program is computation-intensive, as in the case of modexp, as the checks added by multi-specialization must be performed many times. This leads to the biggest degradation of all, peaking at +50% running time.

Combining this increase in running time plus the degradation due to the type change from SIMPLE_INTEGER to PLS_INTEGER, we get a total slowdown of 1.9, which is still smaller than the 2.2 slowdown of the native compilation when changing the used type. This is due to the fact that not all nodes of the AST are affected by this change. For example, the code that is in an if-branch, or in the body of a loop, is never executed with the value NULL because the condition evaluates to false. This means that the condition has multiple specializations and is slower, but once in the body of the loop, only integer values are known to appear. This kind of optimizations and very localized specialization is only possible using runtime specializations, and allow the code to be generic only where it needs to be.

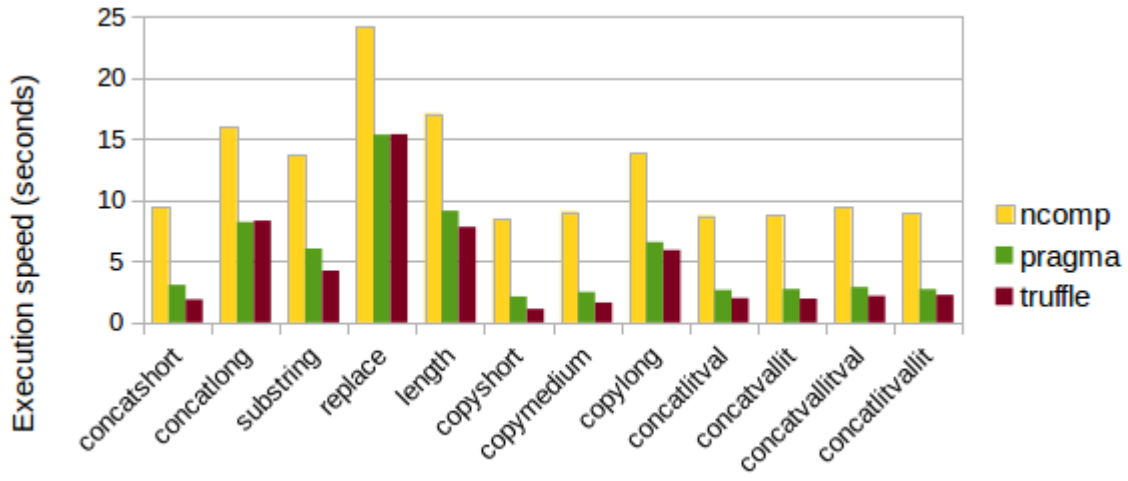

Figure 29: Execution speed for the VARCHAR2 short UDF benchmarks

6.5 VARCHAR2 performance

6.5.1 Comparison with the database

We now evaluate the performance of our VARCHAR2 implementation. The methodology is very similar to the numeric case. We perform micro-benchmarking on SQL queries ran on tables of 10 million entries. The difference will come from the systems compared. When running VARCHAR2-based programs, the difference between the interpreter and the native compilation is rather small. This is due to the nature of string operations on immutable values : they tend to involve a lot of copies. Furthermore, the result must be stored in a specific buffer. Therefore, a copy is always necessary, if only just to return the value.

We start once again with the short UDF case. We wrote a set of very small UDF programs that uses the VARCHAR2 builtins we implemented. The first 5 functions perform a single call to a builtin. Then, the copy benchmarks simply return their input, trigerring a copy into the result buffer. This is usfeul to measure the impact of context switching once again. Finally, the last programs perform common concatenation patterns that can be found in typical PL/SQL programs. Results for these benchmarks is shown in Figure 29.

We can see a steady improvement over almost all the benchmarks. In some cases, the speed of our interpreter is the same as what could be achieved before, but it is never worse. Overall, we get a 25% improvement in execution speed. This result is not as good as the speedup we achieved in the SIMPLE_INTEGER case for UDFs. The main reason for that is the lack of conversion. In the numeric benchmarks, every input value had to be converted. This operation was done using code specialization, which is more efficient than doing it normally. Furthermore, due to the need for a copy of the result into the output buffer, a significant portion of the execution time is dedicated to unavoidable buffer copy, and that for every evaluated system. The cost of such a copy is made explicit in the copyshort, copymedium and copylong benchmarks.

Another interesting thing to note here is the fact that the cheaper context switching Walnut provides us allows us to outperform the database. Otherwise, these benchmarks are too simple to give any room for optimization. For example, in the concatshort case, the operation must take the two input buffers, and copy them both into the output buffer. There is no way around it, and not really anyway to speed that process up. However, we

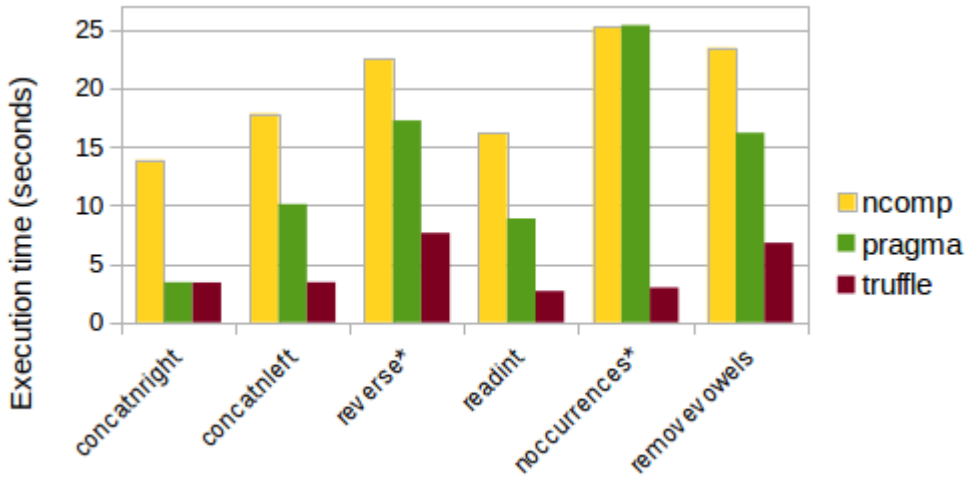

Figure 30: Execution speed of the `VARCHAR2` loop-based programs
**scaled down*

still get a 55% improvement on this benchmark, due to the optimized context switching. Notice that this does not translate to more intensive benchmarks, like concatlong, because the longer time spent copying the buffers minimizes the impact of context switching. On such benchmarks (concatlong and replace), we get no improvement with our solution, but we do not perform worse. This shows that the solution we use reaches the speed of the most simple and bare-metal solution for basic operations.

To get some benefits from our complex solution, we must increase the amount of computation performed. As we did before, we devise another set of benchmarks, more computation-intensive, using loops. We run these programs on the same number of table entries (10 million), and present the results in Figure 30.

The first thing to note here is the difference in behaviours between the concatnright and concatnleft case. These two programs build a string by appending (resp. prepending) n times the same string, where n is an input. In the database systems (both ncomp and pragma), building a string by prepending takes a lot more time then when appending, although the arguments to the functions are the same. This is typical of classic concatenation algorithms, where one would over-allocate the buffer, and keep appending to the current value, like the StringBuilder class in Java. Although this method is very efficient when building strings by appending, prepending becomes very expensive because a new buffer must be allocated every time, and the current value must be copied. Our solution does not suffer from this, because appending and prepeding are done without copy, by updating a data structure.

Another interesting thing to note on these results is the case of removevowels. This benchmark performs multiple calls to the `REPLACE` builtin. Although our implementation of this functionality showed no improvement over the pragma system in the case of a single call, it yields a 2.4x improvement in this benchmark. The reason for it is that removevowels, as its name suggests, removes some letters, i.e. it calls `REPLACE` with a `NULL` replacement. We took the opportunity to optimize specifically this case. One specialized version of our algorithm thus only deals with deletion cases. This makes the algorithm much simpler, and leads to this significant speedup.

Overall, we get greater improvements with the longer-running programs. The average speedup over the best database system (pragma) is 2.7x, with a maximum at 8.5x. This

maximum is achieved for the noccurrences benchmark, which performs a lot of substrings and comparisons. In our system, the former is almost a no-op (it only adds an index to a runtime value), which explains this huge improvement.

6.5.2 Impact of having multiple representations

We just saw that the implementation of VARCHAR2 we provide is rather efficient, as it never performs worse than the database, and usually has significant speedups. We are now interested in knowing what is the reason for this improvement. More specificaly, we wish to validate our design, i.e. show that the faster execution indeed comes from having multiple data representations for string values, with a specialized representation for fast concatenation.

In order to do so, we re-implement efficiently our system with less data representations available. We will compare the following cases:

Only heap values available (1 representation). This requires copy of input values into Java arrays, allocated on the heap. Although this obviously degrades performance, this variant is the simplest. The fact that only one representation is possible simplifies the implementation, and allows for more specific optimizations that would be cumbersome to write otherwise. We use here classic over-allocated buffers for concatenation, making for cheaper append.

Only base values available. This includes heap values and native values (so 2 representations). The burden of input copy is removed, but implementation has more cases and different specializations. Upon concatenation, values are copied into a buffer, which delays the copy of external values.

Full system (3 representations). This is the system described in this project, containing the first two cases plus the linked-list based representation for concatenation.

To compare these different systems, we run them on the same benchmarks as above. To show the relation between these systems, but also with the database implementation, we normalize the results to the time taken by pragma. This ensures that the results from the different, high-varying execution time programs can all be shown in a meaningful way. On top of that, it gives us a simple way to see which implementations perform better or worse than pragma. We display these results in Figure 31.

As expected, the first two systems have some additional overhead compared to our full system. This is mainly due to duplicate copy operations. Even in the simplest case, like concatshort, both the 1- and 2-representations variants must copy the values to be concatenated into a heap buffer, before copying this value out to the return buffer. Although this very simple case seems easy to optimize, it is very hard to make it work in the general case. This would require us to know where the return value comes from, which cannot necessarily be found statically. This becomes a huge cost when copying large strings, leading to poor performances on concatlong. The overhead of the naive approach, i.e. with only heap values, is about +80% on average when compared to the linked-list solution. This is more than enough to make this system worse than the currently-existing solution (i.e. pragma).

The second system also performs worse than our complete design. It does remove a lot of useless copies, for example when performing a LENGTH or SUBSTRING operation. This allows the second variant to approach the performance of the full system. In some cases, they exhibit comparable speeds : susbtring, replace, length ... This is due to the fact that these benchmarks do not use concatenation, which is the main difference between the 2-representation implementation and the final one. For other operators, the specialization path taken by the two versions is the same, because the inputs are always of the same kind

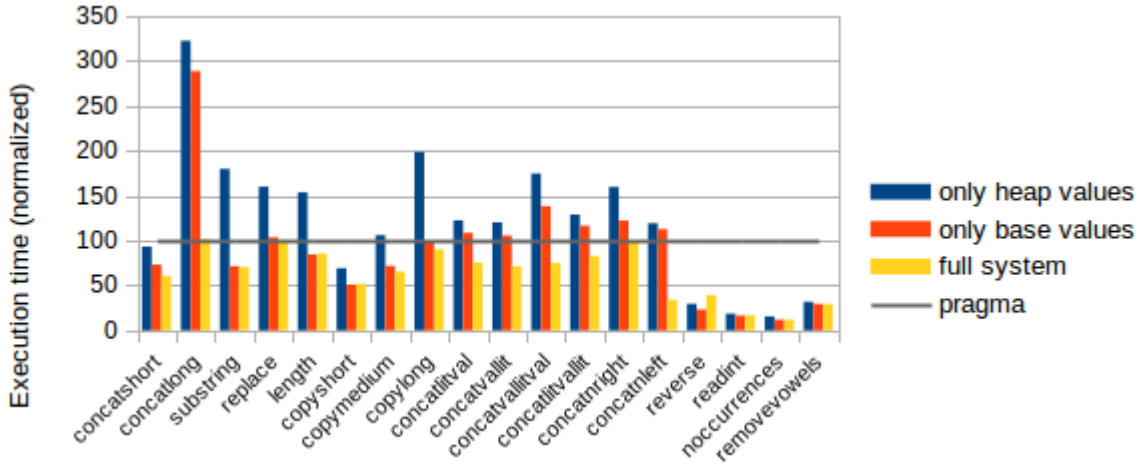

Figure 31: Comparison of the different `VARCHAR2` implementations (lower is better). Execution time is normalized to the time taken by the pragma system (shown as the horizontal line).

(native values). Still, this system is slower than the linked-list approach when it comes to concatenation, because it performs an extra copy into the heap. With this, the overhead averages 37%. In cases where the complete design was already on par with pragma, this is enough to tip the balance and make the 'base values only' version slower.

One can notice that the overheads in the last 3 programs is very small. In the case of the 1-representation case, this is only due to input copy. But the reverse benchmark is the only one that goes the other way around : removing data representations actually improved performance ! This speedup is quite significant : 24 % when using only heap values, and up to 41 % with support for native values. This benchmark makes heavy use of substring and concatenation operators to reverse the input string. Our design being optimized for concatenation, this should not happen. This is due to the function concatenating a lot of single characters strings in a loop. As we saw when discussing the limitations of our solution, the memory overhead for small strings is significant. Thus, most of the time is spent allocating a lot of memory for wrapping a single character. Because these allocations are performed in a loop, they cannot be escape analyzed by Graal, and the memory must be used. In the simpler variants. concatenating one character is very fast, because it only means copying one character. To solve this problem, we could integrate this buffer-based concatenation for this specific case. By profiling the length of the strings being concatenated, it should be possible to detect this bad case, and use buffers instead of lists. This is left for future work.

Overall, we saw our final solution is performing better than the others. This validates our design, showing that it is possible to improve performances using multiple data representations. When no improvement is possible, our system behaves like the current database implementation of `VARCHAR2`, and never worse. Even though it uses a lot of high-level abstractions, which could represent an overhead in simple programs, these are optimized away by the compiler when possible, and leaves us with the simplest implementation possible. On more complex programs, these abstractions showed their benefits when building strings, copying characters only when the result is complete.

7 Future work

Much work can still be done on this project, and this should be a continued effort. We present here some of these future directions, interesting ideas of continued topics, and pointers to solutions that could fix the issues the design chosen still has.

First, for this project to be a complete and correct subset of PL/SQL, a lot of items must be added or fixed. The question still remains open how to prove that our interpreter has exactly the same semantics as the classic execution options for PL/SQL. Answering this question would help acceptation of this solution by potential users. The way it is currently done is through a set of ever-extending unit tests, but this is clearly not enough. More complete testing on full tables queried using SQL could provide more evidence of similar semantics.

Another correctness issue not dealt with is that of concurrency. While we did not provide a mechanism for in-database function call resolution (i.e. a PL/SQL function calling another), adding such a basic feature would require complex and expensive concurrency checks to ensure that the function called still hasn't been modified while the program is running. Although semantics for handling such a case are rather simple in PL/SQL (execution should be aborted by raising an exception), these kind of checks, performed on every call, could prove devastating for the performance of our interpreter.

We did not show in this work how easy interoperability could be in the Graal/Truffle environment, but this would be a very interesting subject. The most obvious language we want to interact with is the SQL execution engine that was implemented on top of Truffle. Using this instead of the classic SQL engine would reduce context switching costs, and allow Graal to perform global optimizations on the whole SQL expression (inlining the PL/SQL UDF call for example). It could be very interesting to try such cooperation with the other languages being added into the database, and ran in the Graal/Truffle environment. A prime example is JavaScript, which is the most advanced project in this direction. Legacy code written in PL/SQL could then be kept, and called directly from new JavaScript programs running in the database, ensuring the same behaviour can be reproduced.

To improve performance further, there is also a lot to do. For example, a full `NUMBER` implementation could be very efficient. Programmers usually tend to avoid this SQL type in computation intensive setups, because of the high cost of operations handling this type. This is all due to the complex memory representation of these, optimized for storage and precision. Using runtime data representation specialization, we should be able to lift this restriction, and use simpler representation when possible. Such work has already been done for the Truffle SQL interpreter, and it would be interesting to integrate this work in a procedural language, with more intensive computations.

To improve on what we implemented so far, we could be using more of the specialized tools Truffle provides. The profiling primitives, not heavily used in this project, could improve the performance by simplifying assumptions made by the compiler. Moreover, we did not really touch in that project the subject of implementation specialization *per se*. We did use type specialization, but it could be interesting to look into very specific code patterns we can identify, and provide an efficient implementation of some operations, only concerned about this case.

Our `VARCHAR2` implementation could be improved as well. Adding more type information into the list representation, adding more representations for specific patterns or richer specializations could be done. Furthermore, to reduce the memory overhead of our solution, classic list techniques could be used. For example, list unrolling is especially simple

to do in runtime systems, with the help of profiling as well, and made even simpler by the use of horizon-based linked lists. Indeed, such a solution could have linked list nodes with a fixed number of entries, and another scope, specific to the nodes, help filling the arrays and keeping the values immutable. This reduces memory consumption and improves data locality.

Finally, we did not look at the memory consumption of the systems during performance evaluation. Only concerned about execution, we should look into the memory footprint of the Graal compiler, which is expected to be higher than the database's solutions. This is due to Truffle making multiple copies of the ran AST (for various reasons), as well as the storage of deoptimization information in the dynamically compiled code. Although not critical for a single user, the memory footprint could end up costing a lot if there are thousands of sessions active at the same time, all running our interpreter. This would limit the deployability of our solution in production systems.

8 Conclusion

In this project, we implemented an efficient AST interpreter for PL/SQL using Graal and Truffle. The use of specialization allowed the code to be compiled and aggressively optimized only for the relevant set of inputs observed at runtime. Using static typing information, we were able to use different specialized data representations for a single data type. This allowed us to use primitive types whenever possible, even in the context of more complex types.

We proposed an implementation of a string data type, tailored to the use and specificities of the PL/SQL language, and showed its improvement over classic solutions in terms of execution time. We showed that using multiple representations for a complex type such as string, and changing it to fit our needs is possible in a runtime environment using dynamic compilation and optimization, and made easier by having a clear abstraction of the original data type, that hides the implementation details.

We demonstrated that using speculative optimizations, dynamic compilation, code and data specialization was able to improve performance of typical data-centric workloads by 2 to 6 times over a basic natively compiled version of the same code. The degradation of the system when faced with a more complete set of inputs was proven to be relatively smooth, and still more efficient than a completely generic interpreter for the same code.

These improvements were achieved by simply writing an AST interpreter comprised of 4105 lines of Java code, showing the power of the Graal and Truffle language environment. We expect the assumptions made on workloads and rarity of `NULL` to hold in real systems, guaranteeing performance improvements that should translate well into production systems.

References

[1] Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks", 1970.

[2] OpenJDK, Graal project.
`http://openjdk.java.net/projects/graal`

[3] Home page of the Walnut project, Oracle Labs.
`https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:15`

[4] T. Würtinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software*, 2013.

[5] Oracle Database PL/SQL Language Reference. Oracle Database version 12c, May 2017. Available at `https://docs.oracle.com/database/121/LNPLS/toc.htm`.

[6] Oracle Database SQL Language Reference. Oracle Database version 12c, March 2017. Available at `https://docs.oracle.com/database/121/SQLRF/toc.htm`.

[7] Charles Wetherell. Freedom, Order, and PL/SQL Optimization. *Oracle white paper*, Dec. 2003.

[8] G. Goos, W. A. Wulf, A. Evans Jr. and K. J. Butler. *DIANA: An Intermediate Language for Ada*. Springer, Berlin, Heidelberg, 1983.

[9] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer and H. Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[10] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, 2013.

[11] G. Duboscq, T. Würthinger and H. Mössenböck. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *Proceedings of the 2014 International Conference on Principes and Practives of Programming on the Java platform: Virtual machines, Languages, and Tools*, 2014.

[12] Truffle source code repository on Github.
`https://github.com/graalvm/graal/tree/master/truffle`

[13] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon and C. Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the Dynamic Languages Symposium*, 2012.

[14] C. Humer, C. Wimmer, C. Wirth, A. Wöß and T. Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, 2014.

[15] Transact-SQL reference manual. Available at `https://docs.microsoft.com/en-us/sql/t-sql/language-reference`

[16] PL/pgSQL - SQL Procedural Language, in *PostgreSQL 9.6.3 Documentation*. Available at `https://www.postgresql.org/docs/9.6/static/plpgsql.gtml`

[17] Dmitry Melnik. Speeding up query execution in PostgreSQL using LLVM JIT compiler. Sept. 2016. Slides available at `https://llvm.org/devmtg/2016-09/slides/Melnik-PostgreSQLLLVM.pdf`

[18] *Ada 2012 Reference Manual: Language and Standard Libraries*, International Standard ISO/IEC 8652:2012(E). Springer, 2013.

[19] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. In *Software - Practice & Experience*, volume 25, issue 12, pages 1315-1330. Dec. 1995.

[20] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. Aug. 2004.

[21] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Jess Ruderman, Edwin Smith, Rick Reitmair, Mohammad R. Haghighat, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM, 2009.

[22] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, Michael Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, School of Computer Science, University of California, Irvine, 2007.

[23] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, pages 294-303, May 2002.

[24] Haichuan Wang, Peng Wu, and David Padua. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[25] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, 2016.

[26] Christian Wimmer and Stefan Brunthaler. ZipPy on truffle: a fast and simple implementation of python. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, Oct. 2013.

[27] Henrique Nazare Santos, Pericles Alves, Igor Costa and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2013.

[28] Craig Chambers. The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. PhD thesis, Computer Science Departement, Stanford University. March 1992.

Annex

We detail in this annex the different benchmarks used, and give the complete set of benchmarking results.

Numeric benchmarks

We start with the numeric benchmarks. The code for the `SIMPLE_INTEGER` and `PLS_INTEGER` programs is exactly the same, except for the type used by the local variables. An example is given in Figure 22. The inputs for all of these functions are `NUMBER` values, randomly generated integers between -1000 and +1000. Therefore, the first operation performed by these functions is to convert the arguments to the correct integer type currently benchmarked. All of these programs are then run on a table comprised of 10 millions such entries.

Description of the numeric UDF benchmarks :

add	Adds the two input numbers
sub	Substracts the two input numbers
mul	Multiplies the two input numbers
add3	Adds the three input numbers
add4	Adds the four input numbers
exp1	Computes the result of $(a - b) \times (a + b)$
scalprod	Performs the scalar product of the vectors (a, b) and (c, d)
wntexp	Performs a complex parenthesised expression mixing both constants and arguments
const	Returns a constant value. This is helpful to measure the overhead of context switching.
constadd	Adds a constant value to the input number
search	Simple `CASE` expression with 6 branches, conditioned on the input values

Description of the numeric stored procedures :

introot	Computes the integer square root of the input value (floor of the real root)
intdiv	Computes the Euclidean quotient of dividing a by b
intmod	Computes the Euclidean remainder of dividing a by b. This is a re-implementation of the `MOD` builtin, which is only defined for `NUMBER` values (requiring an expensive conversion)
gcd	Computes the greatest common divisor (GCD) of a and b
modexp	Computes the modular exponent of the inputs, i.e. $a^b \mod c$

Description of the numeric computation-intensive programs :

countprimes	Counts the number of prime numbers between 1 and 50'000
countfactors	Counts the total number of prime factors every number from 1 to 20'000 has

`VARCHAR2` benchmarks

We explain further the set of benchmarks used to measure the performance of the `VARCHAR2` implementations. These programs are all ran on tables with 10 million randomly generated entries. The length of the strings used depend on the benchmark. We generate three main types of strings :

- Short : length is between 1 and 10
- Medium : length is between 1 and 100
- Long : length is between 1 and 1000

The length of every entry is picked at random, uniformly distributed between the min and the max. The strings generated only contain uppercase letters, to increase the chance of the `REPLACE` benchmarks doing some actual work. We now give details on the every benchmarks, including the lengths of the strings used.

Name	Lengths used	Description
concatshort	short	concatenates the inputs
concatlong	long	concatenates the inputs
substring	long, plus two numbers	call to the `SUBSTR` builtin
replace	long, short, short	call to the `REPLACE` builtin
length	long	call to the `LENGTH` builtin
copyshort	short	returns the input value. This requires a copy of the the input to the output buffer, and helps measure the cost of context switching.
copymedium	medium	returns the input value
copylong	long	returns the input value
concatlitval	medium	returns the concatenation of a fixed literal with the input (common pattern)
concatvallit	medium	returns the concatenation of the input with a fixed literal (common pattern)
concatvallitval	medium	returns the concatenation of an input with a fixed literal and another input (common pattern)
concatlitvallit	medium	returns the concatenation of a fixed literal with the input and another fixed literal (common pattern)

Name	Lengths used	Description
concatnright	medium, short, one number	appends the second input n times to the first input
concatnleft	medium, short, one number	prepends the second input n times to the first input
reverse	medium	returns the input string, read backwards. It does so by appending every input character in reverse order, using `SUBSTR` and concatenation
readint	short numeric string (only digits)	returns the integer encoded by this string's sequence of digits. This is a re-implementation of the `TO_NUMBER` builtin for integers.
noccurrences	long, short	counts the number of times the first input contains the second
removevowels	medium	removes the vowels in the input string. Uses 6 calls to `REPLACE`