Deep Integration of Programming Languages in Databases with Truffle

V.Benzaken G. Castagna L. Daynès K. Nguyen

January 6, 2017

Collaboration Overview

Amount and type of funding requested	239.800 Euros, unrestricted gift
Expected project start date and duration	January 2015. 44 months
External research institution Mailing Address	Université Paris Sud Fondation Paris Sud - SAIC Université Paris-Sud Bâtiment 640 - PUIO 91405 ORSAY cedex
External principal investigator(s) (EPI)	Véronique Benzaken, Giuseppe Castagna, Kim Nguyen veronique.benzaken@u-psud.fr, Giuseppe.Castagna@cnrs.fr, Kim.Nguyen@lri.fr
University gift office representative	Éric Henriet, eric.henriet@u-psud.fr
University administrator to provide banking details	Michaël Martial, michael.martial@u-psud.fr
Oracle principal investigator	Laurent Daynès, Consulting Member of Technical Staff, Oracle Labs, laurent.daynes@oracle.com

1 Project Background and Description

Databases are typically optimized for a particular data model (e.g., relational, semi-structured, graph-based), or a combination of them, and interfaced with a unique query language (SQL, CQL, HiveQL, XQuery, JSoniq, etc.). Database applications, on the other hand, are written in a general purpose programming language that offers developers a large choices of libraries that greatly simplifies presentation to the end-users, communication with other software components, etc. For a variety of architectural reasons, databases applications execute in an environment distinct from that of the databases, i.e., on a client machine (increasingly, a connected mobile device) or in a middle-tier.

This situation causes two main problems that have been the focus of research for several decades:

1. how to better integrate database querying with application programming languages to eliminate impedance mismatch while retaining the full power of the database querying capabilities,

2. how to minimize the traffic, both in terms of number of interactions and volume of data exchanged, between the database and its applications.

With respect to integrating query to programming languages, the industrial landscape is currently dominated by ORM solutions,(Hibernate, Ruby-on-Rail, SQLAlchemy, DJango, Propel, RedBeanPHP, to name the most prominent). These offer frameworks based on popular architectural patterns (active records [], data mappers []) that greatly simplifies application development by wrapping CRUD database operations in type-safe object-oriented interfaces, allowing developers to write code in the host language only. Unfortunately, these solutions favor writing code that iterates over collections of objects representing database records using idioms of the language to perform bulk operation likes filters and joins that would be better done by the database [**?**]. This often results in poor performance as it both increases traffic with the database and can overwhelm the application's memory with very large intermediate results.

To avoid these problems, a number of language-integrated query techniques for embedding queries into general-purpose programming languages have emerged, which seek to reconcile the goals of type-safety, uniform programming idioms, and better capturing of querying intent to optimize interactions with databases. Two directions are being investigated:

1. Use some form of static analysis or type system to identify part of programs that can be turned into queries [**?**, **?**]
2. Extend conventional language with explicit quotation or surface syntax for expressing queries more directly. This approach has been popularized in the industry by Microsoft's LINQ.

LINQ represents a *principled* approach to querying data sources from a programming language. Data sources, or in LINQ's jargon *data providers*, are abstracted away and presented to the programmer by means of a unified API (e.g. in C#, one has access to an `IQueryable` interface that must implement `Select`, `Aggregate`, `Where`, . . . methods). LINQ also supports facilities to extend the syntax of the host language (C#) to add query constructs, whose symbolic representation (their Abstract Syntax Tree or AST) is passed down to the `Select`-like methods. This seemingly allows the programmer to mix expressions (in particular user-defined function calls) from the host language with query predicates. This approach is however too restricted in our opinion.

First, while it is true that the queries and the programming language are integrated from a syntactic stand point, the whole LINQ infrastructure and program logic resides in the language runtime (the .NET VM). Indeed, *data providers* communicate with their backends in a standard RPC style fashion. For instance, the relational backend of LINQ generates SQL queries from the `IQueryable` object, that are sent to a relational database through a standard database driver. Therefore, deep integration of language expressions and queries is but an illusion, and complex queries (featuring user-defined functions) often need several round-trips between the data base and the .NET runtime to exchange intermediate results.

Second, a heavy burden is put on the *data provider* designer, who has to resort to either *(i)* develop a superficial provider, that only implements the most basic query primitives or *(ii)* invest a considerable amount of time, effort and expertise in developing a sophisticated data provider capable of analyzing the syntactic representation of queries and of translating it into one or more request that can be comprehended by the backend. In all cases, sub-expressions from the host language that participate in the query must be translated into an equivalent expression in the programming interface of the data provider (e.g., SQL). This translation may not always be possible, or would require a substantial amount of work, such as, providing an equivalent stored procedure at the database side for all user-defined functions used in the application queries. When this isn't possible, a complex query expression must be split into multiple ones and intermediate results must be materialized at the application side in order to apply the host language's sub-expressions.

This is a trait that LINQ shares with all of the solutions mentioned above, and one that cannot be solved as long as data providers offer a unique querying interface. This research proposal seeks to address the above issues by making the querying interface of data providers effectively multilingual. The goal is to define an intermediate representation of queries (QIR) that is common to application programming languages and data providers of different nature (e.g., Relational DBMS, key-value stores, map-reduce data stores, XML/JSON/RDF and other NoSQL databases,

etc...). Database systems supporting the QIR interface can execute queries requested in QIR form, without requiring applications to translate them first into their main declarative querying interface (e.g., SQL, HiveQL, XQuery).

QIR borrows to the Volcano query evaluation system [**?**], in which data model operations are strictly separated from bulk-operation. The idea behind this clean separation is to make the representation of queries agnostic of any particular data model and operations over data types of that model. Conversely, it allows different query engines to adopt different implementations for data collections and for the algebraic operators that manipulate them, and for the generation of an executable query plan independently of how individual data manipulation are implemented. For this approach to work, query execution engines must be able to execute the data manipulation part of the queries, and, ideally, fuse the code generated for the bulk operators of the physical query plan with the code generated from the representation of data manipulations.

We propose to achieve this using Oracle Lab's stack for building high-performance embeddable virtual machines. The stack relies on three components:

1. the Truffle framework for building high-performance *Abstract Syntax Tree (AST)* interpreters,

2. the Graal dynamic compiler, a new generation just-in-time compiler written in Java and capable of compiling Truffle ASTs using aggressive speculative optimization that can be deoptimized back to the original AST form for latter recompilation when speculations are invalidated.

3. SVM, a substrate of virtual machine services (garbage collection, code cache management, stack walking, deoptimzation, etc.) for building lean embeddable Truffle/Graal virtual machines.

Language developers define the AST nodes according to Truffle's rules for describing the parent-child relationship of the nodes, non-local control flows, runtime replacements of nodes for various profile-driven optimizations of the interpreter, and various directives to inform Graal's about speculations, their invalidation, and other profiling data. The Graal compiler generates at runtime deoptimizable code according to some runtime compilation policy. Truffle is already being used to implement several popular programming languages, namely, JavaScript, Ruby, Python, and R.

Truffle ASTs can be simply rewritten into a low-footprint architecture-neutral form and quickly reconstructed elsewhere for immediate execution, provided availability of the Truffle runtime and AST node factory for the language. Graal is only necessary for aggressive JIT compilation taking advantage of Truffle. Thus, Truffle ASTs can serve as a universal high-level representation of code.

The core of our proposition is to extend Truffle with a factory of QIR nodes that corresponds to comprehensive set of bulk operations with well-defined semantics that can be used to construct an AST representation of queries. QIR nodes resemble higher-order functions that can accept as parameters arbitrary ASTs describing functions or expressions for data manipulation. These data manipulation AST may be generated from any Truffle languages, provided a set of coercion operators for types from the data provider to types of the host language is defined.

The set of proposed QIR nodes must be general enough to represent queries over data of different natures (relational tables, XML or JSON documents, csv files, etc.), that can easily translate in the queried data provider's own representation, and that can be extended as new query operators are defined.

Supporting QIR imposes several requirements to a data provider, namely:

1. it must support execution of truffle ASTs

2. its internal representation of a query must support using Truffle AST to describe user-defined functions and expression over data

3. its query execution engine must be able to convert the data items used in variable-binding expressions involving expressions of the Truffle language before evaluating the expression via its instance of the language's Truffle interpreter.

The first requirement is trivial for data providers that already execute on top of the JVM (e.g., Cassandra, Asterix DB). Other data providers need to build embeddable version of the programming languages they want support for, using SVM. This is currently being prototyped by Oracle Labs for the Oracle 12c RDBMS in the context of the Walnut project.

Addressing the second and third requirements are more involved and requires evolving the query engine of the data provider. We intend to investigate the implications of these requirements in collaboration with Oracle Labs for the Oracle RDBMS, and on our own for a selection of open-source ones.

Figures 1 and 2 shows how the QIR-based architecture we propose differs from the current state of affairs.

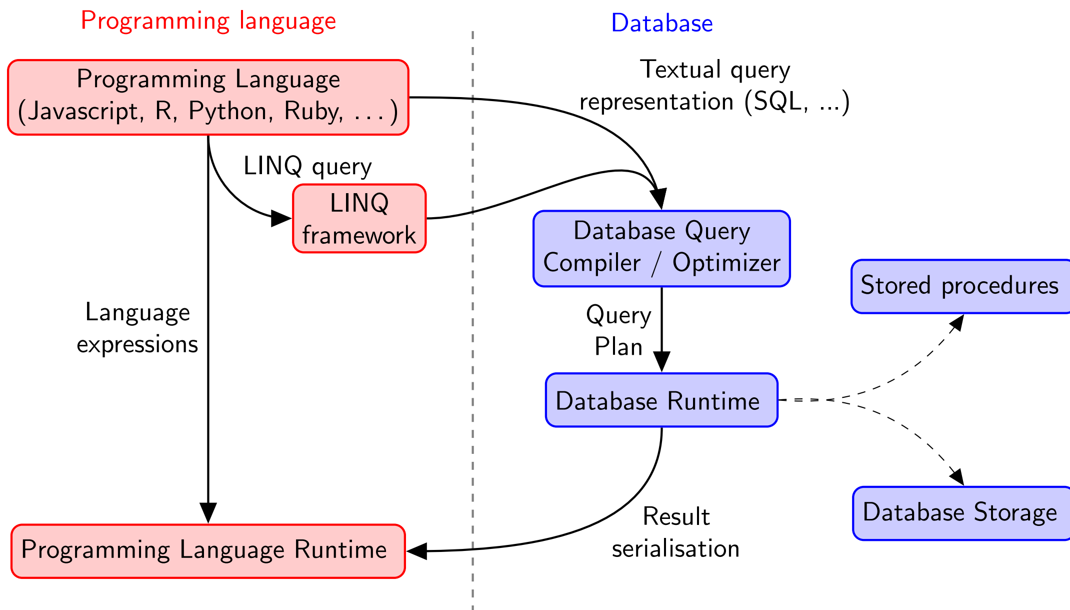

Figure 1: Typical database application architecture.

Database are typically written in some general purpose programming languages (e.g., Ruby, Python, R, etc.) and execute in an environment that is physically *distinct* from that of the DBMS, as show in Fig 1. Interactions with the database are performed via a driver that abstracts the details of the communication protocol with the RDBMS, and offer an API for sending a request described in the declarative language of the DBMS (e.g., SQL), and iterating through the returned results. Various syntactic supports may be provided to reduce the impedance mismatch between the host language and the database's (e.g., ORM frameworks, LINQ), or to minimize the number of interactions. But the interactions with the DBMS remain based on constructing query strings that are sent to the DBMS. If queries relies on user-defined functions, these must also be available at the database side, as stored procedure for instance, to avoid costly materialization of intermediate results at the application side.

The overall software architecture that we aim at developing is described in Figure 2 and would allow the following use case:

1. Application developers write *query intensive* JavaScript program using some language integrated query syntax (e.g., comprehension-based syntax). The queries in the program heavily use JavaScript functions from within queries (e.g., in aggregation, selection or filtering operations).

2. The program is translated into a Truffle AST, featuring special QIR nodes for query primitives;

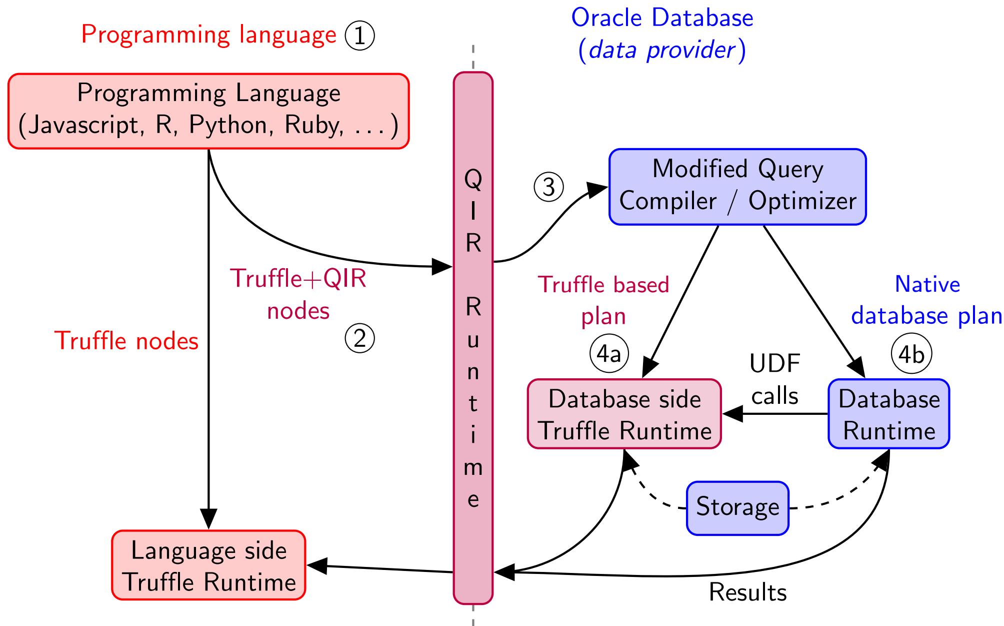

Figure 2: Software architecture of project. The language runtime and the database runtime interact through a QIR Runtime.

3. Parts of the AST (those pertaining to queries) are passed to the database, which first transcript them in its own representation before compiling them into query plans. The database's query compiler is Truffle-ready, meaning that it supports a query representation decorated with Truffle AST expressions and UDF.

4. The whole query plan may itself be compiled into a Truffle program (a), or may call out to an embedded JavaScript Truffle interpreter to execute the JavaScript expression/functions in the query plan (b). The former approach is the prefered one, as it brings the benefits of Graal JIT compilation to the whole query plan.

Note that while this use case is one of the most relevant ones, it is only a particular instance of our approach. Indeed we aim at supporting several dynamic host languages (JavaScript, Python, Ruby...) and several data providers (Oracle DBMS, Java 8 Stream API, Apache Hadoop, ...). In particular, by simply translating QIR nodes into SQL, we can express within our framework an "SQL provider", by translating QIR nodes into SQL queries that can be executed on any relational database. While this is not the prefered use-case for our framework, it can be a convenient fallback mechanism to support database system for which the runtime is not Truffle ready.

The different components of our software stack will be the results of the modular and targeted technical objectives we highlight in the following section.

2 Technical Objectives

Four technical objectives are defined for this project:

1. The definition of a Truffle-compatible Query Intermediate Representation (QIR) that can serve as a common interface between Truffle implementation of various programming languages and data providers.

2. The specification of different ways of embedding queries into host languages and their implementation using QIR. The targeted programming languages will primarily be general-purpose programming language with an existing Truffle implementation, but at least one exemplar genuine database programming language that already provides querying support should be targeted.

3. Several examples of how data providers can support execution of queries described with QIR. The examples shall demonstrate various degree of integration of a data provider with a Truffle runtime.

4. Defining tools for analyzing and, if necessary, rewriting QIR in order to minimize traffic between a QIR client and a QIR data provider.

The following describes in more details how the project will fulfill these objectives.

2.1 Description of Tasks

The work will be organized in four technical Tasks:

Task 1: Definition of the Query Intermediate Representation (QIR)

The core of this research is the definition of a set of expressions/operators to express query operations in a host language. The QIR must balance expressiveness, succinctness and effectiveness:

expressiveness: it must support various querying paradigms and operations: selection, co-grouping, top-k queries, over different data-models (relational, column-based stores, XML/JSON documents, ...);

succinctness: it must provide a minimal set of abstractions that unify similar querying features of different high-level query languages (selection, aggregation, *etc*);

effectiveness: it must retain enough information both to be translated into an efficient query plan of the underlying storage engine, and to recover the original declarative definition of the query.

The definition of QIR will proceed by successive refinements to be defined from the feedback obtained from the other tasks.

From a technical perspective, we will supplement a Truffle based AST with a set of *generic* QIR nodes for the new querying constructs. By doing so we can effectively extend any Truffle based language with query facilities. Note that as already explained, the Truffle/Graal framework concerns itself not only with the abstract syntax of programs, but also their *runtime behaviour* (for instance JIT code specialization and de-optimization when necessary). For QIR nodes, this runtime behaviour will consist in isolating subtrees of the AST that represent queries, decorate them accordingly if necessary with relevant information and pass them to the underlying data provider. Data providers supporting QIR can then translate it into their own representation (see 2.1).

Task 2: Mapping of QIR to Data Providers' logical algebra

One of the goal of the project is not to replace the query optimizers of the targeted data provider, but, on the contrary, to exploit them. Optimizers of mature RDBMS have accumulated years of knowledge that will be difficult to replace. Supporting QIR imposes three requirements on a data provider: the ability to translate QIR to its own representation of queries, without translation into a higher level language; the ability to embed a Truffle virtual machine for the targeted application language, so that expressions from that particular language can be executed; and the ability to keep the Truffle AST that decorates a QIR throughout the query compilation by its optimizer.

The main target for QIR will be the Oracle Database. This will allow, say, a JavaScript programmer to leverage the speed of both Oracle Database (for data access) and Graal (for user defined function execution, and possibly, JIT query compilation). This part will be done by the Oracle partner.

The interface between QIR and the data provider must be sufficiently flexible to allow a thorough palette of compilation targets. In particular, we want to gauge the expressiveness and generality of QIR by targeting data providers for different data models (cloud-based storage, main memory engines, etc.) which include specific and open source query optimizers. This is why we will also consider secondary compilation targets. Typical examples of such targets could be

1. key-value stores to be accesses via Hadoop (Apache) or Coherence (Oracle) or slightly higher level query languages such as the Cassandra Query Language,
2. BigData management systems such as Asterix DB
3. main memory data structures, with particular focus on Java collections and the Java 8 Stream API. This would allow in particular one to quickly prototype data oriented applications that could be evaluated with a unique runtime (the JVM).
4. other open-source relational databases such as MySQL

In order to ease our task we will primary focus on providers implemented in Java.

A problem to be considered later in the development of this task is the transmission of information from the language level to the data provider level. In particular we will have to study how to instrument the implementation of QIR such that Truffle generated statistics can be communicated to the DB query optimizer. For example, a user-defined function may produce a small set of values that can be used to evaluate the selectivity of the filter it is used with. Means to evaluate this at query optimization may substantially impact performance.

Another information that could be used by the query engine is the information about the types inferred at the language level, since this can be used, for instance, to choose a more efficient representation for base types in the database side.

To illustrate, assume a complex query performing integer arithmetic to filter out some results. If one can determine from the Truffle nodes that computation can be carried out using only a smaller integer data type (e.g., `SMALLINT`), then the database engine can choose to represent such numbers with the appropriate data type in intermediary tables (rather than the more generic `INTEGER` or `DECIMAL`) one, resulting in an improvement of both speed and memory footprint for that particular query.

Task 3: Developing querying extension with QIR

QIR can be used to ease extending existing programming languages implemented with Truffle (e.g., JavaScript, Ruby, R, Python) with language integrated query syntax, or to ease the implementation of existing database programming languages on top of Truffle. Although our effort will focus on the integration of QIR into general purpose language, it will be interesting to check whether and to which extent QIR will be able to account for query oriented languages such as Jaql or MongoDB.

For this two different aspects must be taken into account: (1) how to abstract data and (2) how to represent queries.

Data abstraction. We want to provide a clean abstraction to the low level data-store (or logical) view. Again, this integration can be made by either using syntax extensions (DSL approach) or by using constructs available in the language (similar to the Active Records of Ruby, or Object-Relational mappings). It will be an important test to check whether the latter fit our approach so that programs and programmers that use them can smoothly migrate to it.

Query representation. The definition of QIR must be general enough to permit the following three different kinds of integration into a host language.

1. Integration of QIR in the host language by the means of a library API.
2. Addition of a Domain Specific Language (DSL) into an host language via quotation and antiquotation mechanisms
3. Reuse of existing syntax of the host language (with possible tweaks and/or extensions) for seamless integration of QIR.

Let us illustrate each of these three approaches by a concrete example using Python and Ruby as host languages. First we define an higher-order function `inrange`:

```
(* Python *)
def inrange (l,h) :
    return (lambda x : (l < x) & (x < h))

(* Ruby *)
def inrange (l,h) :
    lambda{|x| (l < x) && (x < h)}
```

Then we assume that QIR provides a list comprehension construct that we want to use to express a query parametric in some predicate function `p`. With the first approach we would use the functions, say, `Select`, `Where`, etc, exported by a `QIR` library as follows:

```
(* Python *)
import QIR

db = new QIR.Datasource('...')

def query(base,p):
    return base.Select(lambda x : x.name).Where(lambda x : p(x.age))

result = db.exec(query(db.people,inrange(30,42)))

(* Ruby *)
include QIR

db = Datasource.new('...')

def query(base,p)
      base.Select(lambda{|x| x.name}).Where(lambda{|x| p(x.age)})

result = db.exec(query(db.people,inrange(30,42)))
```

As we see there is not much a difference between the versions in Python and Ruby. A more uniform way to integrate queries is to use quotation (e.g., `{{_}}` to single out expressions of QIR) and antiquotation (e.g., `%` to embed foreign language expressions in QIR syntax) to write queries directly in the syntax of QIR (here, `select_from_where`), which corresponds to the second approach of the list:

```
(* Python *)
def query(p : QIR(Int->Bool)):
    return ({{ select x.name from x in db.people where %p(x.age) }})

result = db.exec(%query(%inrange(30,42)))
```

```
(* there is an implicit coercion from 30 40 in QIR  *)
(* to the corresponding values in the host language *)

(* Ruby *)
def query(p) @__ QIR(Int -> Bool) __@
     {{ select x.name from x in db.people where %p(x.age) }}

result = db.exec(%query(%inrange(30,42)))
```

while the previous approach does not require any particular extension of the syntax, here we see that, besides quotations and anti-quotations, the host language must provide a way to include QIR types, either as a syntactic extension (as for Python) or in source code decorations (in commentaries as for Ruby). Finally the highest level of integration is obtained by reusing as much of the host syntax as possible, as shown by the following code snippets:

```
db = QIR.createengine('...')

(* Python *)
def query(p):
    return [x.name for x in db.people if p(x.age)]

result = db.exec(query(inrange(30,42)))

(* Ruby *)
def query(p)
     (db.people).select{|x| p(x.age)}.collect{|x| x.name}

result = db.exec(query(inrange(30,42)))
```

Here we see that each language uses its own list comprehension syntax to express the query and that the fact that these constructs will actually be translated into QIR ASTs is transparent to the programmer.

Each approach has its own advantages and drawbacks. In a nutshell the approaches are listed from the hardest to the easiest for the language programmer and from the easiest to the hardest for the implementer of the host language.

The API approach is a minimum requirement for a QIR implementation. It requires the minimal effort for the implementer of the host language who wants to include QIR query capabilities in it, but it is the least flexible approach since it may suffer of an impedance mismatch insofar as the existing data structures of the host language may not be flexible enough to fully express all aspects of QIR.

The second approach is more flexible since it essentially uses the QIR types system and thus avoids major impedance mismatch problems, but it also puts more burden to the implementer and the programmer: the former must extend the compiler/interpreter to parse quotations, insert implicit coercions between data and deal with typing aspects; the latter must learn a DSL language to write queries.

Finally the last approach is, in principle, the one that requires the least effort to the programmer, since it mostly reuse existing syntax, but it is the hardest to implement and the least modular since it requires to solve specific problems for each host language. A non exhaustive list of such language specific problems is: define static analyses to recognize code fragments that, after possible rewriting and partial evaluation, can be translated into QIR; determine at which level of the abstract syntax generation (compiler usually use different abstract syntaxes, that they refine during compilation) the switch between host language AST and QIR AST must be performed; handling QIR types and their integration in a possible host type system.

We will explore all these approaches trying not to favor one with respect to the other. Although the third approach is the one that can make our framework to be more easily and rapidly adopted by programmers, we aim at a framework in which the three of them are possible and target a possible gradual adoption of them where, ideally, a language implementer would start integrating QIR in a host language, first by defining an API, then by enriching it by quotation and antiquotation mechanisms and finally by defining all the static machinery needed to translate code fragments without quotation into QIR (either by a source to source translation that would add quotations, or directly into QIR ASTs).

Task 4: Static analyses and logical optimization for QIR

Once the tasks above have reached a fairly stable status, we will start to study the definition of static analyses for QIR, focusing our efforts essentially on three aspects: (*i*) primitive type conversions, (*ii*) predicate inference, and (*iii*) query rewriting/regrouping.

1. Primitive type compatibility and implicit conversions:

   An important problem is to determine when and where to add primitive type conversions and above all in which direction. For instance, consider the following code snippet of the host language:

   ```
   reduce (function (x, y) { return x+y; },0 , collection)
   ```

   that folds the sum operation on a large collection of data that is accessible by some data provider. This is written in the host language. Intuitively at the beginning `0`, `x` and `+` belong to the host language while `y` will be bound to elements in the data provider part. What about `x+y`? Do we want to execute the computation on the language side or on the data provider side? If we decide to delegate this computation to the data provider, then we must convert everything in the data provider representation:

   (a) convert the constant `0` from the language level representation to the database representation

   (b) perform the aggregation on the database side

   (c) convert the result back into a language level data-type

   If instead we want to perform the iteration on the programming language side, then we have to extract in turn each individual element of the collection and translating each of them into a language level data-type before performing the sum.

   While the first approach look at first sight as the most efficient since it requires a predetermined and minimal number of conversions and can exploit the extraction capabilities of the query provider, this must be assessed by taking into account other factors. For instance, if the result of `x+y` is used elsewhere in the host language program, then it may be more efficient to follow the second approach and convert all partial results into a host language data-type and memoize them. Other factors to take into account for such a decision are the cost of primitive conversions, the cost of the operations, and the kind of architecture we are working with (whether the truffle interpreter is integrated in data provider or it is the language host Truffle interpreter that enriched with QIR nodes)

   The analysis may become much more complicated if instead of considering simple programming patterns like the previous one, one wants to provide a principled approach for more complex —and useful— patterns. For instance, consider:

   ```
   reduce (function (x, y) { return g(x+y); } , 0 , collection)
   ```

   where `g` is some function defined in the host language. Where is the threshold that makes conversion go to one side or another?

2. Predicate inference:

   We want to detect by a suitable static analysis the expressions that match predicates of the underlying query engine, since the engine use them to estimate selectivity and generate the corresponding plans. Still, we want to do it so that the programming in the host language is data-model agnostic.

   This problem is related to the previous one: in

   ```
   reduce (function (x, y) { return g(x+y); } , 0 , collection)
   ```

   once we have determined that `g` must be translated to the language of the data side (that is that the coercion on `x+y` goes from the host language to the query engine language), we have to determine whether and how `g` can be expressed by some predicate of the query engine or a combination thereof.

   Several techniques will be experimented to solve the problems of implicit conversion and predicate inference. In particular we will explore the use of specific type systems, of data-flow analyses to explicitly drive the conversions.

3. Query regrouping and rewriting:

   While the previous aspects are meaningful for single queries, it will be interesting to consider programs with multiple queries and perform a global program analysis to determine whether queries can be regrouped and rewritten in order to minimize interactions between a program and the remote data providers. This use of static analysis share some similarities with prior work on query extraction [?, ?], with the difference that we'd ideally want the analysis to be language-agnostic and exploit, if possible, knowledge of the underlying program representation as QIR/Truffle AST. The difficulty is that each language implementation uses it own definition of AST operations and underlying type systems.

2.2 Outcomes

Theoretical outcome. On a theoretical level, the outcome of this project will be the identification of a set of QIR primitives with the characteristics described in Task 1 and allowing a seamless integration with higher level host languages and lower level query engines.

Such a QIR would serve as a theoretical foundation for the formal study of data-oriented languages, which is one of the key research objective of the programming language and database research community alike, and a cornerstone of our academic research projects.

Practical outcome. The practical outcome of the project will consist in the implementation of QIR within the Truffle framework. This implementation will include :

- the integration of query primitive in Truffle-based dynamic languages, initially in the "Simple Language" distributed with Truffle and later on in a full-fledged language such as JavaScript or Python.
- the mapping of QIR primitives on particular storage back-ends with a primary focus on Oracle DBMS, the Java 8 Stream API, and Hadoop column store.

Tools. We also expect that several research results will take the shape of additional tools built on top of our core technology. Such tools may include:

1. static analyzer for single query guarantee (stemming from Task 4)
2. IDE plugins with on-the-fly query generation (stemming from Task 1 and 3). The idea would be to provide a programmer with the (back-end) query that will be generated from his/her JavaScript code for instance

3. IDE plugins for type conversions (stemming from Task 3). Such a plugin would show the programmer what will be the database level type of an expression of the host language and show the coercions that will take place at runtime

The software produced by the external research organization will be licensed under an appropriate free software license such as the GPL v2 license.

2.3 Timetable and resources

The work will span over a period of three and a half years and will require the hiring of a post-doc researcher, of a PhD student and the supervision of three master students. The post-doc position will focus essentially on language aspects.

The goal of the PhD thesis (to be achieved by the end of the project) will be the refinement of QIR (Task 1), the embedding in JavaScript (Task 3), comparison with existing methods such as Active Records, ORM (Task 4), binding with the Oracle Relation Database data layer (Task 2).

The master students will develop particular well-circumscribed aspects of the project.

As described in Section 2.1 the work will be organized in four tasks to bu pursued mostly in parallel but that will be started with 6 months gaps as described next:

Task 1 T0 → T18. From T0 to T6, first draft of QIR, implementation in "simple language"; from T7 refinement of QIR from the feedback of tasks 2 and 3.

Task 2 T7 → T36.

Task 3 T7 → T36

Task 4 T12 → T42

Time-line:

T0-T6 Recruitment and supervision of a master intern. The goal of the internship will be the definition of QIR (Task 1) and a thorough comparison with the current way of performing queries in JavaScript (Task 3).

Recruitment of a post-doc position (18 months). The goal of this recruitment will be the definition of QIR in collaboration with the master student (Task 1), the integration of QIR as a DSL into the "Simple Language" provided with the Truffle distribution (Task 3), and the mapping of QIR into a simple storage to be determined.

Total person*month = 17 (12 hired, 5 permanents)

T7-T12 Start of PhD position.

For the period at issue (T7-T12) we expect to have a first delivery consisting in the implementation of QIR as Truffle nodes and its integration in the "Simple Language" distributed with Truffle. This integration will be tested against a simple main memory data provider, typically in Java 8 by the PhD student (Task 2).

Total persons*months = 17 (12 hired, 5 permanents)

(Note: in case of lack of excellent candidates for the PhD position, the funding will be used to recruit one year of post-doc + 1 year of research engineer.)

T13-T18 Recruitment of a master internship to work in collaboration with the PhD student: start of the work on Java-based data provider (Hadoop/AsterixDB is a likely candidate since its is a platform with a proper query optimizer, it's own algebra, etc., and so we will tackle issues similar to those that will be faced with the Oracle RDBMS).

The PhD student and the post-doc will also study type-systems and quotations/anti-quotations techniques for determining conversions in the host language (primary JavaScript) (Task 3) and develop techniques to implement query regrouping and avoid query cascades (Task 4)

Total person*months = 23 (18 hired, 5 permanents).

T19-T24 PhD position: continue the work on Java-based data provider. Pursue of the work on query regrouping and query cascades (Task 4). Work on predicate inference (Task 3)

Total person*months = 9 (6 hired, 3 permanents)

T25-T36 Validation of the binding with storage engines different from the Oracle data access layer (Task 2); development of Truffle nodes for other languages (typically Python) (Task 3); binding with the Oracle RDBMS. This work will be performed by the PhD student (or the research engineer recruited instead), with the possible help of a master intern.

Total person*month = 22/28 (12/18 hired, 10 permanents)

T37-T42 Logical optimizations of QIR and XML/XQuery deep integration (Task 2,4). To be done by the PhD student with the possible help of a master intern.

Total person*month = 22/28 (12/18 hired, 10 permanents)

2.4 Estimated costs

We estimate that the effort that must be provided from the external research organization will be about 99 months-person. Of these 30 will be provided by the permanent researchers, 36 by a PhD student, 18 by postdoctoral researchers and 15 by internships.

The total funding required to cover the expenses is estimated at about 240.000€ (see Annex for details).

2.5 Expertise of the External investigators

The groups in the two universities defined, developed, and are currently extending and maintaining the language CDuce (`http://www.cduce.org`) a state of the art strongly-typed functional language to manipulate data in XML format. As such the groups have expertise in the theory and implementation of functional languages, type systems, polymorphic languages, and efficient compilation of pattern matching. Relevant to the research of this proposal is that the techniques developed for CDuce have been used to extend the OCaml compiler to manipulate XML data via a quotation/anti-quotation mechanism (OCamlDuce).

The principal investigators have expertise in formalizing query languages for weakly structured data (the so-called, NoSQL languages) [Benzaken et al., 2013], in defining and implementing XML query engines [Diego Arroyuelo et al., 2013, Sebastian Maneth and Kim Nguyễn, 2010], and implementation of database systems and their integration with programming languages [Bancilhon et al., 1988, Benzaken and Delobel, 1990, Benzaken, 1990, Arion et al., 2007]

Furthermore the two groups enjoy the proximity of the persons that are at the origin, developed, and are currently maintaining the Coq system. In particular the VALS group of Paris Sud includes members that are currently working on the Coq formalization of Javascript and, as such, constitute an important reference for the definition and implementation of the language.

3 Relevance to Oracle of Expected Research Outcomes

The research is complementary to Oracle Labs' Walnut project, an effort to modernize Oracle's RDBMS offering. The Walnut project aims at enabling the embedding of Truffle-based virtual machines in the Oracle RDBMS, with the goal of runtime-compiling query plans and support deep embedding of dynamic languages. The latter means that the Oracle RDBMS will be able to support execution of expressions and functions written in a variety of dynamic languages, with JavaScript and Ruby as prime candidates.

The research is also relevant to the research in virtual machine construction conducted by Oracle Labs with the Truffle and Graal projects, as it has the potential to extend their scope of applications to data-centric programming languages.

We expect a number of outcomes from this research:

1. The definition of an AST-based abstract query representation that composes with Truffle ASTs and that is capable of representing queries against various data model.

2. The definition of language integrated query syntax for various languages (JavaScript, R, Ruby, Python), and how to compile these into combined Truffle/QIR ASTs, without modifying the original AST nodes of the languages.

3. An understanding of how a data provider's query engine can implement a QIR interface, i.e., accept queries described with QIR/Truffle and interact with an embedded Truffle virtual machine to execute them. This includes understanding when to convert the representation of data types implemented by the data provider into those implemented by the Truffle-based language, and how data flows between the application and the data provider, etc. This will be based on experimenting with Java-based open-source data providers (e.g., Cassandra) and on close collaboration with Oracle Labs' Walnut project for experimentation with the Oracle RDBMS.

4. An understanding of the impact of this architecture on database application performance. Specifically, execution with a deeply embedded language should outperform solutions based on current practices of using ORM / stored procedures / user-defined functions.

5. Compilation techniques to identify the scope of a query in order to minimize the number of interactions between a client and the data provider taking QIR queries.

The lessons from this research will be learned through the implementation of several prototypes that forms a more concrete outcomes for this research and demonstrate feasibility. We expect to demonstrate prototypes of the following:

1. At least one Java-based data processing engine modified to support QIR and embed a Truffle VM for a Truffle-based language extended with language integrated query syntax.

2. An Oracle 12c prototype capable of executing queries represented as QIR AST for the same Truffle-based language.

References

[Arion et al., 2007] Arion, A., Benzaken, V., Manolescu, I., and Papakonstantinu, Y. (2007). Structured materialized views for XML queries. In *33rd International Conference on Very Large Databases (VLDB 2007)*.

[Bancilhon et al., 1988] Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P., and Velez, F. (1988). The design and implementation of O_2 , an object-oriented database system. In *Object Oriented Database Systems*, Lecture Notes in Computer Science. Springer-Verlag.

[Benzaken, 1990] Benzaken, V. (1990). An Evaluation Model for Clustering Strategies in the O_2 Object-Oriented Database System. In Abiteboul, S. and Kanellakis, P., editors, *International Conference on Dtabase Theory ICDT*, number 470 in Lecture Note in Computer Science, pages 126–140, Paris, France. Springer Verlag.

[Benzaken et al., 2013] Benzaken, V., Castagna, G., Nguyễn, K., and Siméon, J. (2013). Static and dynamic semantics of NoSQL languages. In *POPL '13, 40th ACM Symposium on Principles of Programming Languages*, pages 101–113.

[Benzaken and Delobel, 1990] Benzaken, V. and Delobel, C. (1990). Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In *Proceedings of Persistent Object Systems 4*. Morgan Kaufmann.

[Cheung et al., 2012] Cheung, A., Madden, S., Arden, O., and Myers, A. C. (2012). Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482.

[Cheung et al., 2013] Cheung, A., Solar-Lezama, A., and Madden, S. (2013). Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA. ACM.

[Diego Arroyuelo et al., 2013] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyễn, Jouni Sirén, and Niko Välimäki (2013). Fast in-memory XPath search using compressed indexes. *Software: Practice and Experience*, to appear:to appear.

[Graefe, 1994] Graefe, G. (1994). Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135.

[Sebastian Maneth and Kim Nguyễn, 2010] Sebastian Maneth and Kim Nguyễn (2010). XPath Whole Query Optimization. *PVLDB*, 3(1):882–893.

[Wiedermann and Cook, 2007] Wiedermann, B. and Cook, W. R. (2007). Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 199–210, New York, NY, USA. ACM.

A Costs and ressources

A.1 Estimated costs

The project will need the following human resources:

- 1.5 year postdoc = 80.000€
- 3 years of PhD 110.000€ (or 1 year of post-doc + 1 year engineer)
- 15 months of master internship: 7.000€

Total cost of human resources: 197.000€

To the costs above must be added:

- Costs for travel for the participants (6 persons on 3 years) and equipment for the recruited persons (3 laptops + screen). Overall, 21.000€ (i.e., 1330€per person per year).
- 10% of administration and local overhead costs: 21.800€

TOTAL: 239.800€.

A.2 Contribution of the academic partners

The permanent researchers of academic partners will contribute for a total of 10 months-person per year

- V. Benzaken = 3 months per year (i.e., 50% of research time)
- G. Castagna = 4 months per year (i.e., 33% of research time)
- K. Nguyen = 3 months per year (i.e., 50% of research time)

Furthermore they will provide office facilities, and they will cover other supplementary office and travel expenses in the limit of available credits obtained from other sources for the subject of this research (CISTERA?)