Language-Integrated Queries: a BOLDR Approach

Véronique Benzaken
Université Paris-Sud, France

Giuseppe Castagna
CNRS, Univ Paris Diderot, France

Laurent Daynes
Oracle France, France

Julien Lopez
Université Paris-Sud, France

Kim Nguyễn
Université Paris-Sud, France

Romain Vernoux
ENS Paris-Saclay, France

ABSTRACT

We present BOLDR, a modular framework that enables the evaluation in databases of queries containing application logic and, in particular, user-defined functions. BOLDR also allows the nesting of queries for different databases of possibly different data models. The framework detects the boundaries of queries present in an application, translates them into an intermediate representation together with the relevant language environment, rewrites them in order to avoid query avalanches and to make the most out of database optimizations, and converts the results back to the application. Our experiments show that the techniques we implemented are applicable to real-world database applications, successfully handling a variety of language-integrated queries with good performances.

KEYWORDS

Language-integrated queries, databases, data-centric languages, R

ACM Reference Format:
Véronique Benzaken, Giuseppe Castagna, Laurent Daynes, Julien Lopez, Kim Nguyễn, and Romain Vernoux. 2018. Language-Integrated Queries: a BOLDR Approach. In *Proceedings of The Web Conference 2018 (WWW'18 Companion).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3184558.3185973

1 INTRODUCTION

The increasing need for sophisticated data analysis encourages the use of programming languages that either better fit a specific task (e.g., R or Python for statistical analysis and data mining) or manipulate specific data formats (e.g., JavaScript for JSON). Support for data analysis in data processing platforms cannot follow the pace of innovation sustained by these languages. Therefore, databases are working on supporting these languages: Oracle R Enterprise [18], and PL/R for R; PL/Python [19], Amazon Redshift [2], Hive [3], and SPARK [4] for Python; or MongoDB [14] and Cassandra's CQL [5] for JavaScript. APIs for these embedded languages are low-level, and data is accessed by custom operations, yielding non-portable code. In opposite to this ad hoc approach, language-integrated querying, popularized with Microsoft's LINQ framework [13], proposes to extend programming languages with a querying syntax and to represent external data in the model of the language, thus shielding programmers from having to learn the syntax or data model of databases. To that end, LINQ exposes the language to a set of *standard query operators* that external data providers must implement. However, LINQ suffers from a key limitation: queries can only execute if they can be translated into this set of operators. For instance, the LINQ query

```
db.Employee.Where(x => x.sal >= 2000 * getRate("USD", x.cur))
```

which is intended to return the set of all employees which salary converted in USD is greater than 2000, will throw an error at runtime since LINQ fails to translate the function `getRate` to an equivalent database expression. One solution is to define `getRate` in the database, but this hinders portability and may not be possible at all if the function references runtime values of the language. A more common workaround is to rewrite the code as follows:

```
db.Employee.AsEnumerable()
           .Where(x => x.sal >= 2000 * getRate("USD", x.cur))
```

But this hides huge performance issues: all the data is imported in the runtime of the language, potentially causing important network delays and out-of-memory errors, and the filter is evaluated in main memory thus neglecting all possible database optimizations.

In this work, we introduce BOLDR (**B**reaking boundaries **O**f **L**anguage and **D**ata **R**epresentations), a language-integrated query framework that allows arbitrary expressions from the *host language* (language from which the query comes from) to occur in queries and be evaluated in a database, thus lifting a key limitation of the existing solutions. Additionally, BOLDR is tied neither to a particular combination of database and programming language, nor to querying only one database at a time: for instance, BOLDR allows a NoSQL query targeting a HBase server to be nested in a SQL query targeting a relational database. BOLDR first translates queries into a **Q**uery **I**ntermediate **R**epresentation (or QIR for short), an untyped λ-calculus with data-manipulation builtin operators, then applies a normalization process that may perform a partial evaluation of the QIR expression. This partial evaluation composes distinct queries that may occur separated in the code of the host language into larger queries, thus reducing the communication overhead between the client runtime and the database and allowing databases to perform whole query optimizations. Finally, BOLDR translates and sends the queries to the targeted databases.

Consider again our LINQ query containing the call to `getRate`. In BOLDR, its translation produces a QIR expression according to three different scenarios: (*i*) if `getRate` can be translated into the query language of the targeted database, then the whole expression is translated into a single query expressed in the query language of the targeted database; (*ii*) if `getRate` cannot be entirely translated but contains one or several queries that can be translated, then BOLDR produces the corresponding translated subqueries and sends them to their respective databases, and combines the results at QIR level; (*iii*) if `getRate` cannot be translated at all, then BOLDR creates a query containing the serialized host language abstract syntax tree of `getRate` to be potentially executed on the database side.

WWW'18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3185973

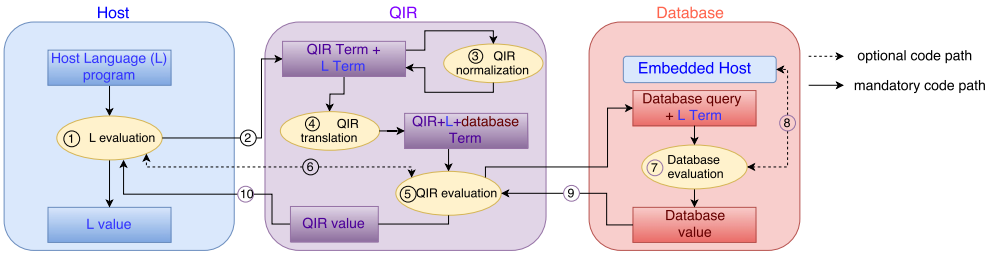


Figure 1: Evaluation of a BOLDR host language program

Our implementation of BOLDR uses Truffle [23], a framework developed by Oracle Labs to implement programming languages. Several features make Truffle appealing to BOLDR: first, Truffle implementations of languages must compile to an executable abstract syntax tree that BOLDR can directly manipulate; second, languages implemented with Truffle can be executed on any JVM, making their addition as an external language effortless in databases written in Java (e.g., Cassandra, HBase, ...), and relatively simple in others such as PostgreSQL. Third, work on one Truffle language can easily be transposed to other Truffle languages.

Our implementation currently supports the *PostgreSQL*, *HBase* and *Hive* databases, as well as *FastR* [17] (Truffle implementation of the R language) and Oracle's *SimpleLanguage* (a dynamic language with syntax and features inspired by JavaScript). The following R program illustrates the key aspects of BOLDR:

```
1  # Exchange rate between rfrom and rto
2  getRate = function(rfrom, rto) {
3    # table change has three columns: cfrom, cto, rate
4    t = tableRef("change", "PostgreSQL")
5    if (rfrom == rto) 1
6      else subset(t, cfrom == rfrom && cto == rto, c(rate))
7  }
8  # Employees earning at least minSalary in the cur currency
9  atLeast = function(minSalary, cur) {
10   # table employee has two columns: name, sal
11   t = tableRef("employee", "PostgreSQL")
12   subset(t, sal >= minSalary * getRate("USD", cur), c(name))
13 }
14 richUSPeople = atLeast(2000, "USD")
15 richEURPeople = atLeast(2000, "EUR")
16 print(executeQuery(richUSPeople))
17 print(executeQuery(richEURPeople))
```

This example is a standard R program with two exceptions: the function `tableRef` (Line 4 and 11) referencing an external source in lieu of creating a data frame (R implementation of tables) from a text file; and the function `executeQuery` (Line 16 and 17) that evaluates a query. We recall that in R, the `c` function creates a vector, the `subset` function filters a table using a predicate, and optionally keeps only the specified columns. The first function `getRate` takes the code of two currencies and queries a table using `subset` to get their exchange rate. The second function `atLeast` takes a minimum salary and a currency code and retrieves the names of the employees earning at least the minimal salary. Since the salary is stored in dollars in the database, the `getRate` function is used to perform the conversion.

In BOLDR, `subset` is overloaded to build an intermediate query representation if applied on an external source reference. The first call to `atLeast(2000, "USD")` builds a query and captures the variables in the local scope. When `executeQuery` is called, then (*i*) the intermediate query is *normalized*, inlining all bound variables with their values; (*ii*) the normalized query is translated into the target database language (here SQL); and (*iii*) the resulting query is evaluated in the database and the results are sent back. After normalization and translation, the query generated for the first call on Line 14 is:

```
SELECT name FROM employee WHERE sal >= 2000 * 1
```

which is optimal, in the sense that a single SQL query is generated. The code generated for the second call is also optimal thanks to the interplay between lazy building of the query and normalization:

```
SELECT name FROM employee WHERE sal >= 2000 *
       (SELECT rate FROM change WHERE rfrom = "USD" AND rto = "EUR")
```

Therefore, BOLDR not only supports user-defined functions (UDFs) in queries, it also merges subqueries together to create fewer and larger queries, thus benefiting from database optimizations and avoiding the "query avalanche" phenomenon [12].

While similar approaches exist (see Section 8 on related work), BOLDR outperforms them on UDFs that cannot be completely translated. For instance, consider:

```
1  getRate = function(rfrom, rto) {
2    cfrom = c("EUR", "EUR", "USD", "USD", "JPY", "JPY")
3    cto = c("USD", "JPY", "EUR", "JPY", "EUR", "USD")
4    rate = c(1.44, 129, 0.88, 114, 0.0077, 0.0088)
5    t = data.frame(cfrom, cto, rate)
6    if (rfrom == rto) 1
7      else subset(t, cfrom == rfrom && cto == rto, c(rate))
8  }
```

This function builds an in-memory data frame using the builtin function `data.frame`. BOLDR cannot translate it to QIR since it calls the underlying runtime, so instead it generates the following query:

```
SELECT name FROM employee
         WHERE sal >= 2000 * R.eval("@...", array("USD", "EUR"))
```

where the string `"@..."` is a reference to a closure for `getRate`.

Mixing different data sources is supported, although less efficiently. For instance, we could refer to an HBase table in the function `getRate`. BOLDR would still be able to evaluate the query by sending a subquery to both the HBase and PostgreSQL databases, and by executing in main memory what could not be translated.

The general flow of query evaluation in BOLDR is described in Figure 1. During the evaluation ① of a host program, QIR terms are lazily accumulated. Their evaluation, when triggered, is delegated to the QIR runtime ② that normalizes ③ the QIR terms to defragment them, then translates ④ them to new QIR terms that contain database language queries (e.g., in SQL). Next, the pieces of these terms are evaluated where they belong, either in main-memory ⑤ or in a database ⑦. "Frozen" host language expressions occurring in these terms are evaluated either by the runtime of the host language that called the QIR evaluation ⑥, or in the runtime embedded in a target database ⑧. Results are then translated from

the database to QIR ⑨, then from QIR to the host language ⑩.

Overview and Contributions. In this work, we introduce BOLDR, a multi-language framework for integrated queries with a unique combination of features such as the possibility of executing user-defined functions in databases, of partially evaluating and merging distinct query fragments, and of defining single queries that operate on data from different data sources. Our technical developments are organized as follows. We first give a formal definition of QIR (Section 3). We then present the translation from QIR to query languages and focus on a translation from QIR to SQL, as well as a type system ensuring that well-typed queries translate into SQL and are avalanche-free (Section 4). We continue by presenting a normalization procedure on the QIR to optimize the translation of a query (Section 5). We next describe the translation from the host language R to QIR (Section 6). Finally, we discuss experimental results (Section 7) of our implementation that supports the languages R and SimpleLanguage and the databases PostgreSQL, HBase and Hive. We show that queries generated by BOLDR perform on a par with hand-written ones, and that UDFs can be efficiently executed in a corresponding runtime embedded in a target database.

2 DEFINITIONS

We give some basic definitions used throughout the presentation.

Definition 2.1 (Host language). A *host language* $\mathcal{H}$ is a 4-tuple $(\mathsf{E}_{\mathcal{H}}, \mathsf{I}_{\mathcal{H}}, \mathsf{V}_{\mathcal{H}}, \overset{\mathcal{H}}{\rightarrow})$ where:

- $\mathsf{E}_{\mathcal{H}}$ is a set of *syntactic expressions*
- $\mathsf{I}_{\mathcal{H}}$ is a set of *variables*, $\mathsf{I}_{\mathcal{H}} \subset \mathsf{E}_{\mathcal{H}}$
- $\mathsf{V}_{\mathcal{H}}$ is a set of *values*
- $\overset{\mathcal{H}}{\rightarrow} : 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{E}_{\mathcal{H}} \rightarrow 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{V}_{\mathcal{H}}$, is the *evaluation function*

We abstract a host language $\mathcal{H}$ by reducing it to its bare components: a syntax given by a set of expressions $\mathsf{E}_{\mathcal{H}}$, a set of variables $\mathsf{I}_{\mathcal{H}}$, and a set of values $\mathsf{V}_{\mathcal{H}}$. Lastly we assume that the semantics of $\mathcal{H}$ is given by a partial evaluation function $\overset{\mathcal{H}}{\rightarrow}$. This function takes an evaluation *environment* (a set of pairs of variables and values, ranged over by σ) and an expression and returns a new environment and a value resulting from the evaluation of the input expression. To integrate a host language we need to be able to manipulate syntactic expressions of the language, inspect and build *environments*, and have access to an *interpreter* for the language.

Definition 2.2 (Database language). A *database language* $\mathcal{D}$ with support for a host language $\mathcal{H}$ is a 4-tuple $(\mathsf{E}_{\mathcal{D}}, \mathsf{V}_{\mathcal{D}}, \mathsf{O}_{\mathcal{D}}, \overset{\mathcal{D}}{\rightarrow})$ where:

- $\mathsf{E}_{\mathcal{D}}$ is a set of *syntactic expressions*
- $\mathsf{V}_{\mathcal{D}}$ is a set of *values*
- $\mathsf{O}_{\mathcal{D}}$ is a set of *supported data operators*
- $\overset{\mathcal{D}}{\rightarrow} : 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{E}_{\mathcal{D}} \rightarrow 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{V}_{\mathcal{D}}$ is the *evaluation function*

Similarly to host languages, we abstract a database language $\mathcal{D}$ as a syntax $\mathsf{E}_{\mathcal{D}}$, a set of values $\mathsf{V}_{\mathcal{D}}$, and an evaluation function $\overset{\mathcal{D}}{\rightarrow}$ which takes an $\mathcal{H}$ environment and a database expression and returns a new $\mathcal{H}$ environment and a database value. Such an evaluation function allows us to abstract the behavior of modern databases that support queries containing foreign function calls. Last, but not least, a database language exposes the set $\mathsf{O}_{\mathcal{D}}$ of data operators it supports, which will play a crucial role in building queries that can be efficiently executed by a database back-end.

3 QUERY INTERMEDIATE REPRESENTATION

3.1 Core calculus

In this section, we define our Query Intermediate Representation, a λ-calculus with recursive functions, constants, basic operations, data structures, data operators, and foreign language expressions.

Definition 3.1. Given a countable set of variables $\mathsf{I}_{\mathtt{QIR}}$, we define the set of QIR *expressions*, denoted by $\mathsf{E}_{\mathtt{QIR}}$ and ranged over by q, as the set of finite productions of the following grammar:

$$\begin{aligned} q ::= {} & x \mid \mathbf{fun}^{x}(x)\mathbf{\rightarrow} q \mid q\ q \mid c \mid op\,(q, \ldots, q) \mid \mathbf{if}\ q\ \mathbf{then}\ q\ \mathbf{else}\ q \\ & \mid \{\, l : q, \ldots, l : q \,\} \mid [\,] \mid q :: q \mid q\ \mathbf{@}\ q \\ & \mid q \cdot l \mid q\ \mathbf{as}\ x :: x\ ?\ q : q \mid o\langle q, \ldots, q \mid q, \ldots, q \rangle \mid \blacksquare_{\mathcal{H}}(\sigma, e) \end{aligned}$$

where $\mathcal{H}$ is a host language.

Besides lambda-terms, QIR expressions include constants (integers, strings, ...), and some builtin operations (arithmetic operations, ...). The data model consists of records and sequences. Records are deconstructed through field projections. Sequences are deconstructed by the *list matching* destructor whose four arguments are: the list to destruct, a pattern that binds the head and the tail of the list to variables, the term to evaluate (with the bound variables in scope) when the list is not empty, and the term to return when the list is empty. The new additions to these mundane constructs are *database operators* and *host language expressions*. A database operator $o\langle q_1 \ldots, q_n \mid q'_1, \ldots, q'_m \rangle$ is similar to the notion of operator in the relational algebra. Its arguments are divided in two groups: the q_i expressions are called *configurations* and influence the behavior of the operator; the q'_i expressions are the sub-collections that are operated on. Finally, a host expression $\blacksquare_{\mathcal{H}}(\sigma, e)$ is an opaque construct that contains an evaluation environment σ and an expression e of the host language $\mathcal{H}$. We use the following syntactic shortcuts:

- $[\, q_1, \ldots, q_n \,]$ stands for $q_1 :: \ldots :: q_n :: [\,]$
- $\mathbf{fun}^{f}(x_1, \ldots, x_n)\mathbf{\rightarrow} q$ stands for $\mathbf{fun}^{f}(x_1)\mathbf{\rightarrow}(\ldots(\mathbf{fun}^{f}(x_n)\mathbf{\rightarrow} q))$
- $q\ (q_1, \ldots, q_n)$ stands for $(\ldots(q\ q_1)\ldots)\ q_n$

Functions can be defined recursively by using the recursion variable that indexes the **fun** keyword, that we omit when useless.

Definition 3.2 (Reduction rules). Let $\rightarrow^{\delta} \subset \mathsf{E}_{\mathtt{QIR}} \times \mathsf{E}_{\mathtt{QIR}}$ be a reduction relation for basic operators and $\rightarrow \subset \mathsf{E}_{\mathtt{QIR}} \times \mathsf{E}_{\mathtt{QIR}}$ be the reduction relation defined by:

$$\begin{aligned} (\mathbf{fun}^{f}(x)\mathbf{\rightarrow} q_1)\ q_2 &\rightarrow q_1\{f/\mathbf{fun}^{f}(x)\mathbf{\rightarrow} q_1,\ x/q_2\} \\ \mathbf{if\ true\ then}\ q_1\ \mathbf{else}\ q_2 &\rightarrow q_1 \\ \mathbf{if\ false\ then}\ q_1\ \mathbf{else}\ q_2 &\rightarrow q_2 \\ \{\ldots, l : q, \ldots\} \cdot l &\rightarrow q \\ [\,]\ \mathbf{as}\ x :: y\ ?\ q_{list} : q_{empty} &\rightarrow q_{empty} \\ q_{head} :: q_{tail}\ \mathbf{as}\ x :: y\ ?\ q_{list} : q_{empty} &\rightarrow q_{list}\{x/q_{head},\ y/q_{tail}\} \end{aligned}$$

$$\begin{aligned} [\,]\ \mathbf{@}\ q &\rightarrow q & (q_1 :: q_2)\ \mathbf{@}\ q_3 &\rightarrow q_1 :: (q_2\ \mathbf{@}\ q_3) \\ q\ \mathbf{@}\ [\,] &\rightarrow q & (q_1 :: q_2)\ \mathbf{@}\ q_3 &\rightarrow q_1 :: (q_2\ \mathbf{@}\ q_3) \end{aligned}$$

where $q\{x_1/q_1, \ldots, x_n/q_n\}$ denotes the standard capture avoiding substitution. We define the reduction relation of QIR expressions as the context closure of the relation $\rightarrow^{\delta} \cup \rightarrow$.

Crucially, embedded host expressions as well as database operator applications whose arguments are all reduced are irreducible.

3.2 Extended semantics

We next define how to interface host languages and databases with QIR. We introduce the notion of *driver*, a set of functions that translate values from one world to another.

Definition 3.3 (Language driver). Let $\mathcal{H}$ be a host language. A *language driver* for $\mathcal{H}$ is a 3-tuple $({}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{H}}, {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}})$ of total functions such that:

- ${}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}} : 2^{\mathsf{I}_{\mathcal{H}} \times \mathsf{V}_{\mathcal{H}}} \times \mathsf{E}_{\mathcal{H}} \to \mathsf{E}_{\mathsf{QIR}} \cup \{\Omega\}$ takes an $\mathcal{H}$ environment and an $\mathcal{H}$ expression and translates the expression into QIR
- $\overrightarrow{\mathsf{VAL}}^{\mathcal{H}} : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathcal{H}} \cup \{\Omega\}$ translates QIR values to $\mathcal{H}$ values
- ${}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}} : \mathsf{V}_{\mathcal{H}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ translates $\mathcal{H}$ values to QIR values

where the special value Ω denotes a failure to translate.

Definition 3.4. (Database driver) Let $\mathcal{D}$ be a database language. A *database driver* for $\mathcal{D}$ is a 3-tuple $(\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{D}}, {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}})$ of total functions such that:

- $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}} : \mathsf{E}_{\mathsf{QIR}} \to \mathsf{E}_{\mathcal{D}} \cup \{\Omega\}$ translates a QIR expression into $\mathcal{D}$
- $\overrightarrow{\mathsf{VAL}}^{\mathcal{D}} : \mathsf{V}_{\mathsf{QIR}} \to \mathsf{V}_{\mathcal{D}} \cup \{\Omega\}$ translates QIR values to $\mathcal{D}$ values
- ${}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}} : \mathsf{V}_{\mathcal{D}} \to \mathsf{V}_{\mathsf{QIR}} \cup \{\Omega\}$ translates $\mathcal{D}$ values to QIR values

where the special value Ω denotes a failure to translate.

We are now equipped to define the semantics of QIR terms, extended to host expressions and database operators.

Definition 3.5 (Extended QIR semantics). Let $\mathcal{H}$ be a host language, $({}^{\mathcal{H}}\overrightarrow{\mathsf{EXP}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{H}}, {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}})$ a driver for $\mathcal{H}$, $\mathcal{D}$ a database language, and $(\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}, \overrightarrow{\mathsf{VAL}}^{\mathcal{D}}, {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}})$ a driver for $\mathcal{D}$. We define the extended semantics $\sigma, q \twoheadrightarrow \sigma', q'$ of QIR by the following set of rules:

$$\frac{q \to q'}{\sigma, q \twoheadrightarrow \sigma, q'} \qquad \frac{\sigma \cup \sigma', e \xrightarrow{\mathcal{H}} \sigma'', w}{\sigma, \blacksquare_{\mathcal{H}}(\sigma', e) \twoheadrightarrow \sigma'', {}^{\mathcal{H}}\overrightarrow{\mathsf{VAL}}(w)}$$

$$\frac{\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle) = e \quad \sigma, e \xrightarrow{\mathcal{D}} \sigma', w \quad o \in \mathsf{O}_{\mathcal{D}} \quad e \neq \Omega \quad {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(w) \neq \Omega}{\sigma, o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \twoheadrightarrow \sigma', {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(w)}$$

Since QIR is an intermediate language from a host language to a database language, the evaluation of QIR terms will always be initiated from the host language runtime. It is therefore natural for the extended semantics to evaluate a QIR term in a given host language environment. If this QIR term is neither a database operator nor a host language expression, then the simple semantics of Definition 3.2 is used to evaluate the term, otherwise the extended semantics of Definition 3.5 is used. Host expressions are evaluated using the evaluation relation of the host language in the environment formed by the union of the current running environment and the captured environment. This allows us to simulate the behavior of most dynamic languages (in particular R, Python, and JavaScript) that allow a function to reference an undefined global variable as long as it is defined when the function is called. Last, but not least, the evaluation of a database operator consists in *(i)* finding a database language that supports this operator, *(ii)* use the database driver for that language to translate the QIR term into a native query, *(iii)* use the evaluation function of the database to evaluate the query, and *(iv)* translate the results back into QIR.

At this stage, we have defined a perfectly viable Query Intermediate Representation in the form of a λ-calculus extended with data operators. We next address the two following problems:

(1) How to create database drivers in practice?
(2) How to avoid query avalanches as much as possible?

4 DATABASE TRANSLATION

In this section, we describe how a database driver can define a translation from QIR to a database language. This translation must be able to translate QIR expressions into equivalent efficient queries of a database language, and handle QIR expressions in which subterms target different databases. Additionally, it must be seamlessly extendable with new database drivers. To that end, we separate this translation in two phases: a *generic* translation that determines the targeted query language for all subterms of a QIR expression, and a *specific* translation that makes use of database drivers.

4.1 Generic translation

The goal of the generic translation is to produce a QIR expression where as many subterms as possible have been translated into native database queries. Ideally, we want the whole QIR expression to be translated into a single database query, but this is not always possible and, in that case, parts of the expression have to be evaluated in the client side (where the QIR runtime resides). The QIR evaluator therefore relies on two components. First, a "fallback" implementation of QIR operators using the QIR itself, that we dub MEM for in-memory evaluation. MEM is a trivial database language for which the translations to and from the QIR are the identity function, and that supports the operators `Filter`, `Project`, and `Join` defined as plain QIR recursive functions. The full definition of MEM is straightforward and given in Section A of our appendix [6]. Second, to allow the QIR evaluator to send queries to a database and translate the results back into QIR values, we assume that for each supported database language $\mathcal{D} \in \mathbb{D}$, we have a basic QIR operator, $\texttt{eval}^{\mathcal{D}}$ defined as:

$$\frac{\sigma, e \xrightarrow{\mathcal{D}} \sigma', v}{\sigma, \texttt{eval}^{\mathcal{D}}(e) \twoheadrightarrow \sigma', {}^{\mathcal{D}}\overrightarrow{\mathsf{VAL}}(v)}$$

Notice that in the case of the MEM language, the operator $\texttt{eval}^{\mathsf{MEM}}$ is simply the reduction of a QIR term.

The generic translation is given by the judgment $q \rightsquigarrow e, \mathcal{D}$ where $q \in \mathsf{E}_{\mathsf{QIR}}$ and $e \in \mathsf{E}_{\mathcal{D}} \cup \{\Omega\}$, which means a QIR expression q can either be rewritten into an expression e of the language $\mathsf{E}_{\mathcal{D}}$ of the database $\mathcal{D}$, or fail when $e = \Omega$. An excerpt of the set of inference rules used to derive this judgment is given in Figure 2. Rule (db-op) states that given a database operator, if there exists a database $\mathcal{D}$ distinct from MEM such that all data arguments can be translated into expressions of $\mathsf{E}_{\mathcal{D}}$, then if the specific translation $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}$ called on the operator yields a fully translated $\mathsf{E}_{\mathcal{D}}$ expression e, then e is returned as a translation in $\mathsf{E}_{\mathcal{D}}$. This rule may fail in two cases: the data arguments of the operator could be translated to more than one database language; or the specific translation for $\mathsf{E}_{\mathcal{D}}$ could yield an error Ω even if all data arguments of the operator have been successfully translated into expressions of the same language $\mathsf{E}_{\mathcal{D}}$, for instance, when the operator is not supported by $\mathcal{D}$, or when the specific translation of a configuration q_i fails. If the operator o at issue is one of the supported operators of MEM, then both cases are handled by the rule (mem-op): each translated subexpression e_i is wrapped in a call to the $\texttt{eval}^{\mathcal{D}_i}$ operator and o is evaluated with its MEM semantics. All the other rules are bureaucratic and propagate the translation recursively to subterms.

$$\text{(db-op)}\quad \frac{q'_i \rightsquigarrow e_i, \mathcal{D}, i \in 1..m \quad \overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle) = e \quad \mathcal{D} \in \mathbb{D} \setminus \mathsf{MEM} \quad e \neq \Omega}{o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \rightsquigarrow e, \mathcal{D}}$$

$$\text{(fun)}\quad \frac{q \rightsquigarrow e, \mathcal{D}}{\mathbf{fun}^f(x)\rightarrow q \rightsquigarrow \mathbf{fun}^f(x)\rightarrow \mathsf{eval}^{\mathcal{D}}(e), \mathsf{MEM}}$$

$$\text{(mem-op)}\quad \frac{q'_i \rightsquigarrow e_i, \mathcal{D}^i, \in 1..m \quad \begin{array}{c} o \in \mathsf{O}_{\mathsf{MEM}} \\ e_i \neq \Omega \end{array}}{o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle \rightsquigarrow \overrightarrow{\mathsf{EXP}}^{\mathsf{MEM}}(o\langle q_1, \ldots, q_n \mid \mathsf{eval}^{\mathcal{D}^1}(e_1), \ldots, \mathsf{eval}^{\mathcal{D}^m}(e_m)\rangle), \mathsf{MEM}}$$

$$\text{(app)}\quad \frac{q_1 \rightsquigarrow e_1, \mathcal{D}^1 \quad q_2 \rightsquigarrow e_2, \mathcal{D}^2}{q_1\ q_2 \rightsquigarrow (\mathsf{eval}^{\mathcal{D}^1}(e_1))\ (\mathsf{eval}^{\mathcal{D}^2}(e_2)), \mathsf{MEM}}$$

Figure 2: Some rules of the generic translation

4.2 Specific translation: SQL

We document how to define specific translations using SQL as an example of a database language. QIR to SQL is an important translation as it allows BOLDR to target most relational databases and some distributed databases such as Hive or Cassandra. We assume that the set of values for SQL only contains basic constants (strings, numbers, Booleans, …) and tables. The set of expressions $\mathsf{E}_{\mathsf{SQL}}$ is the set of syntactically valid SQL queries [1]. The set of supported operators $\mathsf{O}_{\mathsf{SQL}}$ we consider is { `Filter`, `Project`, `Join`, `From`, `GroupBy`, `Sort` }. Due to space constraints, we describe these operators and the full translation from QIR to SQL in Section A and B of our appendix [6]. The translation from QIR to SQL is mostly straightforward. However, ensuring that it does not fail is challenging. Indeed, SQL is *not* Turing complete and relies on a flat data model: a SQL query should only deal with sequences of records whose fields have basic types. Another important aspect of this translation is to avoid query avalanche by translating as many QIR expressions as possible.

We obtain these strong guarantees using an ad hoc SQL type system for QIR terms described in Figure 3. This type system is straightforward, but in accordance with the semantics of SQL we require applications of basic operators and conditional expressions to take as arguments and return expressions that have basic types B, and data operators to take as sources flat record lists. We also use a rule to type a flat record list as a base type since SQL automatically extracts the contents of a table containing only one value (one line of one column). For instance, `(SELECT 1)+ 1` is allowed and returns `2`.

Note that we *do not* require the host language to be statically typed. Given a QIR term q of type T in the SQL type system, we ensure that the reduction relation of Definition 3.2 terminates on q and yields a term q' that has type T, and that if q is in normal form, then the generic translation of Figure 2 yields a single, syntactically correct SQL expression (using the translation of Section B of our appendix [6]).

We restrict $\mathsf{E}_{\mathsf{QIR}}$ to non-recursive functions and by removing untranslatable terms (such as list destructors) as well as host expressions since we limit ourselves to pure queries, and by restricting data operators to `Project`, `From`, `Filter`, `Join`, `GroupBy`, and `Sort`. What we obtain is a simply typed λ-calculus extended with records and sequences without recursive functions, which entails strong normalization. We also state an expected subject reduction theorem

Theorem 4.1 (Subject reduction). *Let $q \in \mathsf{E}_{\mathsf{QIR}}$ and Γ an environment from QIR variables to QIR types. If $\Gamma \vdash q : T$, and $q \rightarrow q'$, then $\Gamma \vdash q' : T$.*

and are now equipped to state our soundness of translation theorem

Theorem 4.2 (Soundness of translation). *Let $q \in \mathsf{E}_{\mathsf{QIR}}$ such that $\emptyset \vdash q : T$, $q \rightarrow^* v$, and v is in normal form. If $T \equiv B$ or $T \equiv R$ or $T \equiv R$ **list** then $v \rightsquigarrow \mathsf{s}, SQL$.*

Proofs of these theorems are detailed in Section B of our appendix [6] in which we show that typable QIR terms have particular normal forms imposed by their type that can be translated into SQL expressions.

5 QIR HEURISTIC NORMALIZATION

Our guarantees only hold for a QIR query targeting one database supporting SQL. However, a QIR term may mix several databases or use features that escape the hypotheses of Theorem 4.2. In particular, outside these hypotheses, we cannot guarantee the termination of the normalization. We are therefore stuck between two unsatisfactory options: either (*i*) trying to normalize the term (to fully reduce all applications) and yield the best possible term w.r.t. query translation but risk diverging, or (*ii*) translate the term as-is at the risk of introducing query avalanches. We tackle this problem with a heuristic normalization procedure that tries to reduce QIR terms enough to produce a good translation by combining subqueries.

To that end, we define a measure of "good" QIR terms, and ask that each reduction step taken yields a term with a smaller measure. To formally define this measure, we first introduce a few concepts.

Definition 5.1 (Compatible data operator application). Let $\mathbb{D}$ be the set of database languages. A QIR data operator $o\langle q_1, \ldots, q_n \mid q'_1, \ldots, q'_m\rangle$ is a *compatible operator application* if and only if:
$\exists \mathcal{D} \in \mathbb{D}, e_1, \ldots, e_m \in \mathsf{E}_{\mathcal{D}}$ s.t. $\overrightarrow{\mathsf{EXP}}^{\mathcal{D}}(o, q_1, \ldots, q_n, e_1, \ldots, e_m) \neq \Omega$

Intuitively, a compatible data operator application is one where the configuration arguments are in a form that is accepted by the specific translation of the database language $\mathcal{D}$. We now define the related notion of *fragment*.

Definition 5.2 (Fragment). A fragment F is a subterm of a QIR term q such that $q = C[T(q_1, \ldots, q_{i-1}, F[e_1, \ldots, e_n], q_{i+1}, \ldots, q_j)]$ where C is a one-hole context made of arbitrary expressions; T is a non-compatible j-ary expression; $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_j$ and F are the children of T; F is an n-hole context made only of compatible operators applications of the same database language $\mathcal{D}$; and all $e_1, \ldots, e_n$ have head expressions that are not compatible.

Figure 4 gives a graphical representation of a fragment. We can now define a measure of "good" QIR terms.

$$B ::= \texttt{string} \mid \texttt{int} \mid \texttt{bool} \mid \ldots$$
$$T ::= B \mid T \to T \mid T\ \textbf{list} \mid \{l : T, \ldots, l : T\}$$
$$R ::= \{l : B, \ldots, l : B\}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : T_1 \vdash q : T_2}{\Gamma \vdash \textbf{fun}(x) \to q : T_1 \to T_2} \qquad \frac{\Gamma \vdash q_1 : T_1 \to T_2 \quad \Gamma \vdash q_2 : T_1}{\Gamma \vdash q_1\ q_2 : T_2} \qquad \frac{}{\Gamma \vdash c : \texttt{typeof}(c)}$$

$$\frac{\Gamma \vdash q : T}{\Gamma \vdash q :: [\,] : T\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : T \quad \Gamma \vdash q_2 : T\ \textbf{list}}{\Gamma \vdash q_1 :: q_2 : T\ \textbf{list}}\ (q_2 \neq [\,]) \qquad \frac{\Gamma \vdash q_1 : T\ \textbf{list} \quad \Gamma \vdash q_2 : T\ \textbf{list}}{\Gamma \vdash q_1\ \textbf{@}\ q_2 : T\ \textbf{list}}$$

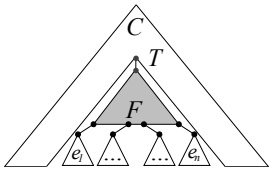
$$\frac{\Gamma \vdash op : B_1 \to \ldots \to B_n \to B \quad \Gamma \vdash b_i : B_i \quad i \in 1..n}{\Gamma \vdash op(b_1, \ldots, b_n) : B} \qquad \frac{\Gamma \vdash b_1 : \texttt{bool} \quad \Gamma \vdash b_2 : B \quad \Gamma \vdash b_3 : B}{\Gamma \vdash \textbf{if}\ b_1\ \textbf{then}\ b_2\ \textbf{else}\ b_3 : B} \qquad \frac{\Gamma \vdash q_i : T_i \quad i \in 1..n}{\Gamma \vdash \{\, l_1 : q_1, \ldots, l_n : q_n \,\} : \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\frac{\Gamma \vdash q_1 : R_2 \to R_1\ \Gamma \vdash q_2 : R_2\ \textbf{list}}{\Gamma \vdash \texttt{Project}\langle q_1 \mid q_2 \rangle : R_1\ \textbf{list}} \qquad \frac{\Gamma \vdash n : \texttt{string}}{\Gamma \vdash \texttt{From}\langle n \rangle : R\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : R \to \texttt{bool} \quad \Gamma \vdash q_2 : R\ \textbf{list}}{\Gamma \vdash \texttt{Filter}\langle q_1 \mid q_2 \rangle : R\ \textbf{list}}$$

$$\frac{\begin{array}{l}\Gamma \vdash q_1 : R_3 \to R_4 \to R_1 \quad \Gamma \vdash q_3 : R_3\ \textbf{list} \\ \Gamma \vdash q_2 : R_3 \to R_4 \to \texttt{bool}\ \Gamma \vdash q_4 : R_4\ \textbf{list}\end{array}}{\Gamma \vdash \texttt{Join}\langle q_1, q_2 \mid q_3, q_4 \rangle : R_1\ \textbf{list}}$$

$$\frac{\Gamma \vdash q_1 : R_3 \to R_1\ \textbf{list} \quad \Gamma \vdash q_2 : R_1\ \textbf{list} \to R_2 \quad \Gamma \vdash q_3 : R_3\ \textbf{list}}{\Gamma \vdash \texttt{GroupBy}\langle q_1, q_2 \mid q_3 \rangle : R_2\ \textbf{list}} \qquad \frac{\Gamma \vdash q_1 : R_2 \to R_1 \quad \Gamma \vdash q_2 : R_2\ \textbf{list}}{\Gamma \vdash \texttt{Sort}\langle q_1 \mid q_2 \rangle : R_2\ \textbf{list}} \qquad \frac{\Gamma \vdash q : \{l : B, \ldots\}\ \textbf{list}}{\Gamma \vdash q : B} \qquad \frac{\Gamma \vdash q : \{l : T, \ldots\}}{\Gamma \vdash q \cdot l : T}$$

Figure 3: QIR type system for SQL



C: surrounding context

F: fragment

T: non-compatible expression

$e_1, \ldots, e_n$: non-compatible expressions

Figure 4: A fragment within a larger QIR term

Definition 5.3 (measure). Let $q \in \mathsf{E}_{\mathsf{QIR}}$ be a QIR expression, we define the measure of q as the pair

$$M(q) = (\mathrm{Op}(q) - \mathrm{Comp}(q), \mathrm{Frag}(q))$$

where $\mathrm{Op}(q)$ is the number of occurrences of data operators in q, $\mathrm{Comp}(q)$ is the number of occurrences of *compatible* data operator applications in q and $\mathrm{Frag}(q)$ is the number of fragments in q. The order associated with M is the lexicographic order on pairs.

This measure works as follows. During a step of reduction of a term q into a term q', q' is considered a better term either if the number of operators decreases, or if q' possesses more occurrences of *compatible* operator applications, meaning less cycles between QIR and the databases, or lastly, if the number of data operators does not change but the number of fragments decreases, meaning that some data operators were combined into a larger fragment.

Our heuristic-based normalization procedure uses this measure as a guide through the reduction of a QIR term: it applies all possible combinations of reduction steps to the term as long as its measure decreases after a number of steps fixed by heuristic. This allows us to generate a more efficient translation while ensuring termination.

Some practical choices impact the effectiveness of the QIR normalization such as choosing which reduction rule to apply at each step (e.g., choosing those with more arguments), or which maximum number of steps to use. Extensive experiments for both points are detailed in a technical report[22]. In particular, we measure that the normalization represents a negligible fraction of the execution time of the whole process compared to tasks such as parsing, or exchanges on the network with databases.

6 FROM A HOST LANGUAGE TO QIR

In this section, we outline how to interface a general-purpose programming language with BOLDR. As explained in Section 1, our aim is to allow programmers to write queries using the constructs of the language they already master. Therefore, instead of extending the syntax of the language, we extend its runtime by reusing existing functionalities, in particular by overloading existing functions.

We use the programming language R as example of a host language to show how to implement a language driver. The full details of our treatment to R can be found in Section C of our appendix [6]. R programs include first-class functions; side effects ("=" being the assignment operator as well as the variable definition operator); sequences of expressions separated by ";" or a newline; structured data types such as vectors and tables with named columns (called *data frames* in R's lingo); and static scoping as it is usually implemented in dynamic languages (e.g., as in Python or JavaScript) where identifiers that are not in the current static scope are assumed to be *global* identifiers even if they are undefined when the scope is created. For instance, the R program:

```
f = function (x) { x + y }; y = 3; z = f(2);
```

is well-defined and stores `5` in `z` (but calling `f` before defining `y` yields an error). We next define the core syntax of R.

Definition 6.1. The set E_R of expressions (e) and values (v) of R are generated by the following grammars:

$$\begin{array}{rcl} e & ::= & c \mid \mathsf{x} \mid \mathsf{function}(x, \ldots, x)\{e\} \mid e(e, \ldots, e) \mid \mathsf{op}\ e\ \ldots\ e \\ & & \mid \mathsf{x} = e \mid e; e \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \\ v & ::= & c \mid \mathsf{function}_\sigma(\mathsf{x}, \ldots, \mathsf{x})\ \{e\} \mid \mathsf{c}(v, \ldots, v) \end{array}$$

where c represents constants, $\mathsf{x} \in \mathsf{I}_\mathsf{R}$, and $\sigma \in 2^{\mathsf{I}_\mathsf{R} \times \mathsf{V}_\mathsf{R}}$ is the environment of the closure.

We recall that, in R, $\mathsf{c}(e_1, \ldots, e_n)$ builds a vector. Definition 6.1 only defines expressions that can be translated to QIR. Expressions not listed in the definition are translated into host expression nodes.

We now highlight how data frames are manipulated in standard R. As mentioned in Section 1, the `subset` function filters a data frame:

```
12  subset(t, sal >= minSalary * getRate("USD", cur), c(name))
```

This function returns the data frame given as first argument, filtered by the predicate given as second argument, and restricted to the columns listed in the third argument. Note that before resolving its second and third arguments, and for every row of the first argument, `subset` binds the values of each column of the row to a variable of the corresponding name. This is why in our example the variables `sal` and `name` occur free: they represent columns of the data frame `t`.

The join between two data frames is implemented with the function `merge`. We recall that the join operation returns the set of all combinations of rows in two tables that satisfy a given predicate.

To integrate R with BOLDR, we define two builtin functions:

- `tableRef` takes the name of a table and the name of the database the table belongs to, and returns a reference to the table
- `executeQuery` takes a QIR expression, closes it by binding its free variables to the translation to QIR of their value from the current R environment, sends it to the QIR runtime for evaluation, and translate the results into R values

We also extend the set of values V_R:

$$v ::= \ldots \mid \texttt{tableRef}(v, \ldots, v) \mid q_\sigma$$

where q_σ are *QIR closure values* representing queries associated with the R environment σ used at their definition.

The functions `subset` and `merge` are overloaded to call the translation ${}^{R}\overrightarrow{EXP}$ on themselves if their first argument is a reference to a database table created by `tableRef`, yielding a QIR term q to which the current scope is affixed, creating a QIR closure q_σ. Free variables in q_σ that are not in σ are global identifiers whose bindings are to be resolved when q_σ is executed using `executeQuery`.

We now illustrate the whole process on the introductory example of Section 1.

Evaluation of the query expression: When an expression recognized as a query is evaluated, it is translated to QIR (using Definition C.2 in Section C of our appendix [6]). In the introductory example, the function call

```
14 richUSPeople = atLeast(2000, "USD")
```

triggers the evaluation of the function `atLeast`:

```
9  atLeast = function(minSalary, cur) {
10   # table employee has two columns: name, sal
11   t = tableRef("employee", "PostgreSQL")
12   subset(t, sal >= minSalary * getRate("USD", cur), c(name))
13 }
```

in which the function `subset` (Line 12) is evaluated with a table reference as first argument, and is therefore translated into a QIR expression. `richUSPeople` is then bound to the QIR closure value:

$$\begin{array}{l}\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\,name\colon t\cdot name\,\}\mid\\ \texttt{Filter}\langle\textbf{fun}(e)\rightarrow e\cdot sal\geq\ minSalary*(getRate\ \texttt{"USD"}\ cur)\rangle\mid\\ \texttt{From}\langle employee\rangle\rangle\rangle\\ \{minSalary\mapsto 2000,\ getRate\mapsto\text{function}_\sigma(\texttt{rfrom},\texttt{rto})\{\ldots\},cur\mapsto\texttt{"USD"}\}\end{array}$$

Query execution: A QIR closure is executed using the function `executeQuery`. In our example, this happens at Line 16 and 17:

```
16 print(executeQuery(richUSPeople))
17 print(executeQuery(richEURPeople))
```

`executeQuery` then resolves each free variable by applying them to the translation to QIR of their value in the R environment:

$$\begin{array}{l}(\textbf{fun}(getRate)\rightarrow\\ \quad(\textbf{fun}(minSalary,\ cur)\rightarrow\\ \quad\quad\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\,name\colon t\cdot name\,\}\mid\\ \quad\quad\texttt{Filter}\langle\textbf{fun}(e)\rightarrow\geq\ (e\cdot sal,*(minSalary,\ getRate\ \texttt{"USD"}\ cur))\mid\\ \quad\quad\texttt{From}\langle employee\rangle\rangle\rangle\\ \quad)(2000,\ \texttt{"USD"})\\ )(\textbf{fun}(rfrom,\ rto)\rightarrow\ldots)\end{array}$$

Next, the QIR runtime is called, and the query is normalized to:

$$\begin{array}{l}\texttt{Project}\langle\textbf{fun}(t)\rightarrow\{\,name\colon t\cdot name\,\}\mid\\ \texttt{Filter}\langle\textbf{fun}(e)\rightarrow\geq\ (e\cdot sal,2000)\mid\\ \texttt{From}\langle employee\rangle\rangle\rangle\end{array}$$

then translated to SQL as:

```
SELECT T.name AS name FROM (
  SELECT * FROM (SELECT * FROM employee) AS E WHERE E.sal >= 2000
) AS T
```

This query is sent to PostgreSQL, and the results are translated back to QIR using ${}^{\text{PostgreSQL}}\overrightarrow{VAL}$, then to R using $\overrightarrow{VAL}^{\mathcal{R}}$.

7 IMPLEMENTATION AND RESULTS

Implementation. BOLDR consists of QIR, host languages, and databases. To evaluate our approach, we implemented the full stack, with R and SimpleLanguage as host languages and PostgreSQL, HBase and Hive as databases. Table 1 gives the numbers of lines of Java code for each component to gauge the relative development effort needed to interface a host language or a database to BOLDR. All developments are done in Java using the Truffle framework.

Component	l.o.c.	Remark
FastR / SimpleLanguage	*173000 / 12000*	not part of the framework
Detection of queries (in R and SL)	600	modification of builtins/operators
R to QIR / SL to QIR	750 / 1000	the translation of Section 6
QIR	4000	norm/generic translation/evaluator
QIR to SQL / HBase language	500 / 400	the translation $\overset{\text{SQL}}{\rightsquigarrow}$ / $\overset{\text{HBase}}{\rightsquigarrow}$
PostgreSQL / Hbase / Hive binding	150 / 100 / 100	low-level interface

Table 1: BOLDR components and their sizes in lines of code.

As expected, the bulk of our development lies in the QIR (its definition and normalization) which is completely shared between all languages and database backends. Compared to its 4000 l.o.c., the development cost of languages or database drivers, including translations to and from QIR is modest (between 700 and 1000 l.o.c.).

Even though our main focus is on Truffle-based languages, on which we have full control over their interpreters, all our requirements are also met by the introspection capabilities of modern dynamic languages. For instance, in R, the `environment` function returns the environment affixed to a closure as a modifiable R value, the `body` function returns the body of a closure as a manipulable abstract syntax tree, and the `formals` function returns the modifiable names of the arguments of a function. These introspection capabilities could be used to achieve an even more seamless integration.

Experiments. The results of our evaluation[1] are reported in Table 2. Queries named `TPCH-`n are SQL queries taken from the TPC-H performance benchmark [21]. These queries feature joins, nested queries, grouping, ordering, and various arithmetic subexpressions. Table 2.**A** and 2.**B** illustrate how our approach fare against hand-written SQL queries. Each row reports the expected cost (in disk page fetches as reported by the `EXPLAIN ANALYZE` commands) as well as the actual execution time on a 1GB dataset. Row SQL represents the hand-written SQL queries, Row SQL+UDFs represents the same SQL queries where some subexpressions are expressed as function calls of stored functions written in PL/SQL. Row R represents the SQL queries generated by BOLDR from equivalent R expressions, and Row R+UDFs represents the same SQL queries as in Row SQL+UDFs generated by BOLDR from equivalent R expressions with R UDFs. Lastly, for R+■, we added untranslatable subexpressions kept as host language nodes to impose a call to the database

[1]The test machine was a PC with Ubuntu 16.04.2 LTS, kernel 4.4.0-83, with the latest master from the Truffle/Graal framework and PostgreSQL 9.5, Hive 2.1.1, and HBase 1.2.6 all with default parameters.

A	TPCH-1		TPCH-2		TPCH-3		TPCH-4		TPCH-5		TPCH-6		TPCH-9		TPCH-10		TPCH-11		TPCH-12		TPCH-13		TPCH-14		TPCH-15		TPCH-16		TPCH-18		TPCH-19	
	10^3 page fetches	time (s)																														
SQL	424	9.44	99	0.42	353	0.99	161	0.49	200	0.59	248	1.03	336	5.54	270	1.72	65	0.33	320	1.61	258	2.07	217	1.05	412	2.06	45	0.94	1462	4.33	306	1.31
SQL+UDFs	1753	15.50	655	0.43	426	2.21	424	0.79	617	1.69	1719	1.90	677	5.34	869	2.90	43	∞	1798	1.95	493	3.20	1766	5.26	207∗	∞	133	206.11	1118∗	∞	216	1.25
R	424	9.37	52	0.66	359	0.97	49939	0.53	200	0.76	248	1.02	300	3.20	272	1.73	65	0.31	334	1.37	258	2.03	217	0.97	412	2.00	39	0.41	2069	4.45	338	1.54
R+UDFs	424	9.51	52	0.68	359	0.95	49939	0.53	200	0.71	248	0.95	300	3.09	272	1.85	65	0.31	334	1.41	258	1.99	217	0.95	412	1.95	39	0.40	2069	4.35	338	1.58

B	TPCH-1		TPCH-3		TPCH-5		Ex. 1	
SQL+UDFs	1753	15.50	426	2.21	617	1.69	0.9	0.05
R+■	1720	37.56	97	2.01	471	8.99	0.9	0.11

C (in s)	Query 1	Query 2	Query 3
Hive	1.24	6.27	6.27
Hive+R	1.25	6.31	6.33

D (in s)	PostgreSQL (atLeast)	HBase (atLeast)	Hive (atLeast)
PostgreSQL (getRate)	0.34	1.47	1.17
HBase (getRate)	1.44	1.33	2.07
Hive (getRate)	0.74	1.78	0.66

∞: evaluation took more than 5 minutes. ∗ wrong cost estimation due to complex PL/SQL function

Table 2: Evaluation of BOLDR's performances

embedded R runtime. The results show that we can successfully match the performances of Row SQL with Row R, and that BOLDR outperforms PostgreSQL in Row R+UDFs against Row SQL+UDFs. This last result comes from the fact that PostgreSQL is not always able to inline function calls, even for simple functions written in PL/SQL. In stark contrast, no overhead is introduced for a SQL query generated from an R program, since the normalization is able to inline function calls properly, yielding a query as efficient as a hand-written one. As an example, the TPCH-15 query was written in R+UDFs as:

```
supplier = tableRef("supplier", "PostgreSQL", "postg.conf", "tpch")
revenue = tableRef("revenue", "PostgreSQL", "postg.conf", "tpch")
max_rev = function() max(subset(revenue, TRUE, c(total_revenue)))
q = subset(merge(supplier, revenue, function(x, y) x$s_suppkey ==
     y$supplier_no),
         total_revenue == max_rev(),
         c(s_suppkey, s_name, s_address, s_phone, total_revenue)
        )[order(s_suppkey), ]
print(executeQuery(q))
```

BOLDR was able to inline this query, whereas the equivalent in SQL+UDFs could not be inlined by the optimizer of PostgreSQL.

Table 2.**B** illustrates the overhead of calling the host language evaluator from PostgreSQL by comparing the cost of a *non-inlined* pure PL/SQL function with the cost of the same function embedded in a host expression within the query. While it incurs a high overhead, it remains reasonable even for expensive queries (such as TPCH-1) compared to the cost of network delays that would happen otherwise since host expressions represent expressions that are impossible to inline or to translate in the database language.

Table 2.**C** illustrates the overhead of calling the host language evaluator from Hive against a pure inlined Hive query. For instance

```
SELECT * FROM movie WHERE year > 1974 ORDER BY title
```

against

```
SELECT * FROM movie WHERE R.APPLY('@...', array(year))ORDER BY title
```

where `'@...'` is the serialization of an R closure, and `R.APPLY` is a function we defined that applies an R closure to an array of values from Hive (including the necessary translations between Hive, QIR, and R). The results are that with one (Query 1/2) or two (Query 3) calls to the external language runtime, the overhead is negligible compared to the execution of the query in Map/Reduce.

Table 2.**D** gives the performances of queries mixing two data sources between a PostgreSQL, a HBase, and a Hive database. We executed the example in the Introduction and varied the data sources for the functions `getRate` and `atLeast`. In the current implementation, a join between tables from different databases is performed on the client side (see our future work in the Conclusion), therefore the queries in which the two functions target the same database perform better, since they are evaluated in a unique database implying less network delays and less work on the client side.

8 RELATED WORK

The work in the literature closest to BOLDR is T-LINQ and P-LINQ by Cheney et al. [7] which subsumes previous work on LINQ and Links and gives a comprehensive "practical theory of language integrated queries". In particular, it gives the strongest results to date for a language-integrated queries framework. Among their contributions stand out: (*i*) a quotation language (a λ-calculus with list comprehensions) used to express queries in a host language, (*ii*) a normalization procedure ensuring that the translation of a query cannot cause a query avalanche, (*iii*) a type system which guarantees that well-typed queries can be normalized, (*iv*) a general recipe to implement language-integrated queries and (*v*) a practical implementation that outperforms Microsoft's LINQ. Some parts of our work are strikingly similar: our intermediate representation is a λ-calculus using reduction as a normalization procedure. However, our work diverges radically from their approach because we target a different kind of host languages. T-LINQ requires a pure host language, with quotation and anti-quotation support and a type-system. Also, T-LINQ only supports one (type of) database per query and a limited set of operators (essentially, selection, projection, and join, expressed as comprehensions). While definitely possible, extending T-LINQ with other operators (e.g., "group by") or other data models (e.g., graph databases) seems challenging since their normalization procedure hard-codes in several places the semantics of SQL. The host languages we target do not lend themselves as easily to formal treatment, as they are highly dynamic, untyped, and impure programming languages. We designed BOLDR to be target databases agnostic, and to be easily extendable to support new languages and databases. We also endeavored to lessen the work of driver implementers (adding support for a new language or database) through the use of embedded host language expressions, which take advantage of the capability of modern databases to execute foreign code. This contrasts with LINQ where adding new back-ends is known to be a difficult task [10]. Lastly, we obtained formal results corresponding to those of T/P-LINQ by grafting a specific SQL type system on our framework.

QIR is not the first intermediate language of its kind. While LINQ proposes the most used intermediate query representation, recent work by Ong et al. [16] introduced SQL++, an intermediary query representation whose goal is to subsume SQL and NoSQL. In this work, a carefully chosen set of operators is shown to be sufficient to express relational queries as well as NoSQL queries (e.g., queries over JSON databases). Each operator supports configuration options to account for the subtle differences in semantics for distinct query languages and data models (treatment of the special value `NULL`,

semantics of basic operators such as equality, ...). In opposite, we chose to let the database expose the operators it supports in a driver.

Grust et al. [12] present an alternative compilation scheme for LINQ, where SQL and XML queries are compiled into an intermediate *table algebra* expression that can be efficiently executed in any modern relational database. While this algebra supports diverse querying primitives, it is designed to specifically target SQL databases, making it unfit for other back-ends.

Our current implementation of BOLDR is at an early stage and, as such, it suffers several shortcomings. Some are already addressed in existing literature. First, since we target dynamic programming languages, some forms of error cannot be detected until query evaluation. This problem has been widely studied and, besides T-LINQ, works such as SML# [15] or ScalaDB [11] use the static type system of the language to ensure the absence of a large class of runtime errors in generated queries. Second, our treatment of effects is rather crude. Local side effects, such as updating mutable references scoped inside a query, work as expected while observable effects, such as reading from a file on host machine memory, is unspecified behavior. The work of Cook and Wiedermann [8] shows how client-side effects can be re-ordered and split apart from queries. Third, at the moment, when two subqueries target different databases, their aggregation is done in the QIR runtime. Costa Seco et al. [9] present a language which allows manipulation of data coming from different sources, abstracting their nature and localization. A drawback of their work is the limitation in the set of expressions that can be handled. Our use of arbitrary host expressions would allow us to circumvent this problem.

9 CONCLUSION AND FUTURE WORK

We presented BOLDR, a framework that allows programming languages to express complex queries containing application logic such as user-defined functions. These queries can target any source of data as long as it is interfaced with the framework, more precisely, with our intermediate language QIR. We provided methods for programming languages and databases to interface with QIR, as well as an implementation of the framework and interfaces for R, SimpleLanguage, PostgreSQL, HBase, and Hive. We described how QIR reduces and partially evaluates queries in order to take the most of database optimizations, and showed that BOLDR generates queries performing on a par with hand-written SQL queries.

Future work includes the creation of a domain-specific language to define translations from QIR to database languages, leaving the implementation details to the language itself, with the associated gains of speed, clarity, and concision. Currently, queries targeting more than one data sources are partially executed in the host language runtime. We plan to determine when such queries could be executed efficiently in one of the targeted data sources instead. For instance, in a join between two distinct data sources, it could be more efficient to send data from one data source to the other that will complete the join. ORMs and LINQ can type queries since they know the type of the data source. BOLDR cannot do it yet since QIR queries may contain dynamic code, and we do not want to type-check the whole host language. While we cannot foresee any general solution, we believe that to exploit any type information available we could use *gradual typing* [20], a recent technique blending static and dynamic typing in the same language. In particular, we would be able to use type information from database schemas (when available) to infer types for the queries.

REFERENCES

[1] 2016. ISO/IEC 9075-2:2016, Information technology-Database languages–SQL–Part 2: Foundation (SQL/Foundation). (2016).
[2] Amazon 2017. Python Language Support for UDFs. (2017). http://docs.aws.amazon.com/redshift/latest/dg/udf-python-language-support.html
[3] Apache 2017. Hive Manual - MapReduce scripts. (2017). https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Transform
[4] Apache 2017. PySpark documentation - pyspark.sql.functions. (2017). http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html
[5] Apache 2017. User Defined Functions in Cassandra 3.0. (2017). http://www.datastax.com/dev/blog/user-defined-functions-in-cassandra-3-0
[6] BOLDR 2018. Online appendix. (2018). https://www.lri.fr/~lopez/www.pdf
[7] J. Cheney, S. Lindley, and P. Wadler. 2013. A Practical Theory of Language-Integrated Query. In *ICFP 2013*. ACM, New York, NY, USA, 403–416.
[8] William R. Cook and Ben Wiedermann. 2011. Remote Batch Invocation for SQL Databases. In *Database Programming Languages - DBPL 201, 13th International Symposium, Seattle, Washington, USA, August 29, 2011. Proceedings*. http://www.cs.cornell.edu/conferences/dbpl2011/papers/dbpl11-cook.pdf
[9] João Costa Seco, Hugo Lourenço, and Paulo Ferreira. 2015. A Common Data Manipulation Language for Nested Data in Heterogeneous Environments. In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/2815072.2815074
[10] O. Eini. 2011. The Pain of Implementing LINQ Providers. *Commun. ACM* 54, 8 (Aug. 2011), 55–61.
[11] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. 2010. Extending Scala with Database Query Capability. *Journal of Object Technology* 9, 4 (2010), 45–68.
[12] Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. *PVLDB* 3, 1 (2010), 162–172. http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R14.pdf
[13] Microsoft 2017. LINQ (Language-Integrated Query). (2017). https://msdn.microsoft.com/en-us/library/bb397926.aspx
[14] MongoDB 2017. MongoDB User Manual - Server-side JavaScript. (2017). https://docs.mongodb.com/manual/core/server-side-javascript/
[15] A. Ohori and K. Ueno. 2011. Making standard ML a practical database programming language. In *ICFP*. ACM, New York, NY, USA, 307–319.
[16] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). http://arxiv.org/abs/1405.3631
[17] Oracle 2017. FastR. (2017). https://github.com/graalvm/fastr
[18] Oracle 2017. Oracle R Enterprise. (2017). http://www.oracle.com/technetwork/database/database-technologies/r
[19] PostgreSQL 2017. PL/Python - Python Procedural Language. (2017). https://www.postgresql.org/docs/9.5/static/plpython.html
[20] J. G. Siek and W. Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*. University of Chicago TR-2006-06, Chicago, USA, 81–92.
[21] TPC. 2017. The TPC-H benchmark. (2017). http://www.tpc.org/tpch/
[22] Romain Vernoux. 2016. Design of an intermediate representation for query languages. *CoRR* abs/1607.04197 (2016). arXiv:1607.04197 http://arxiv.org/abs/1607.04197
[23] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to Rule Them All. In *Onward! 2013 Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, New York, NY, USA, 187–204.