A Common Data Manipulation Language for Nested Data in Heterogeneous Environments

João Costa Seco
NOVA LINCS – Universidade Nova de Lisboa, Portugal

Hugo Lourenço Paulo Ferreira
OutSystems SA, Portugal

Abstract

One key aspect of data-centric applications is the manipulation of persistent data repositories, which is moving fast from querying a centralized relational database to the ad-hoc combination of constellations of data sources.

Query languages are being typefuly integrated in host, general purpose, languages in order to increase reasoning and optimizing capabilities of interpreters and compilers. However, not much is being done to integrate and orchestrate different and separate sources of data.

We present a common data manipulation language, that abstracts the nature and localization of the data-sources. We define its semantics and a type directed compilation, query optimization, and query orchestration mechanism to be used in development tools for heterogeneous environments. We provide type safety and language integration.

Our approach is also suitable for an interactive query construction environment by rich user interfaces that provide immediate feedback on data manipulation operations. This approach is currently the base for the data layer of a development platform for mobile and web applications.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages–Query languages; D.3.3 [*Programming Languages*]: Language Constructs and Features–Data types and structures

Keywords programming languages, data query languages, distributed and heterogeneous queries, type systems

1. Introduction

The state of the art on development of data-centric web, cloud, and mobile applications, is highly based on the use of frameworks, tools, languages and abstractions, specially designed to hide many development and runtime details. One of the key aspects is the safe and easy manipulation of persistent data repositories, usually performed with the help of abstractions like object mappings (e.g. Java JPA), or specialized query languages like Microsoft LINQ.

Obvious benefits are obtained by typefuly integrating query languages in the host programming languages, thus increasing the validation and optimizing power of interpreters and compilers [6, 9, 11, 20]. However, the data manipulation paradigm is moving fast from querying a single data repository, to combining data coming from a constellation of data sources. Heterogeneous queries are pervasive, in scenarios like medical databases and search engines, web service orchestrations, mobile applications, and web or cloud applications that enrich their interfaces with remote web services. Such queries are usually accomplished with ad-hoc code, that is many times inefficient and error prone.

An urgent need arises for development platforms that integrate and query different and separate data sources, in a typeful and seamless way. The wide range of skills needed to query a relational database, efficiently combine the results with a web-service response, and then produce a map-reduce algorithm to join and filter the results in a NoSQL database, is not part of the skill-set of the average developer. Moreover, such an approach contrasts with the data integration efforts of hiding different sources behind a common interface in a very expressive, but predefined way (cf. [13]). This paper introduces a model for a common data manipulation language for heterogeneous data-centric environments, and a compilation method based on type and localization information of data-sources. We define a model to generate specialized and distributed querying code for each (remote) data source, and the corresponding in-memory post-processing code. We model each kind of database system (relational or NoSQL), parameterized data repository (web services), or in-memory data, by a set of capabilities (e.g. to join collections, group by arbitrary expressions, nest results, filter), that guide the way operations are split between locations. Languages like Microsoft LINQ do allow for several kinds of data sources to be involved in a query, but, in this case, the default execution includes fetching all data first and then combine the pieces in memory. Our model decentralizes parts of a query, and is extensible with optimizing execution strategies like [12].

Our model supports the construction and combination of nested collections [5, 8], it is designed from first principles, targeting a general model of data sources, from relational data to nested collections. We introduce a novel language operation whose semantics is the in-place modification of nested data (given a tree-like path, cf. XPath [5, 7]). This operation can either be applied as a in-memory step or be re-written during the code generation process, and incorporated into the target query code, to be executed remotely. This operation is particularly useful in supporting the visual counterpart of this model, that supports the incremental and interactive construction of nested queries with immediate feedback on results. Our approach is the base for an industrial grade development platform for mobile and web applications, the OutSystems Platform [17], where different kinds of data-sources can be used in a typed way, and where the data manipulation language provides type safety and language integration.

Expressions	e, c	$::=$	x
		$\mid$	$\lambda x.e$
		$\mid$	$e\ e$
		$\mid$	$\langle \overline{a = e} \rangle$
		$\mid$	$e \oplus e$
		$\mid$	$e.a$
		$\mid$	$\emptyset$
		$\mid$	$[e]$
		$\mid$	$e \uplus e$
		$\mid$	$\mathsf{exec}\ x = e\ \mathsf{in}\ e$
		$\mid$	q
		$\mid$	num
		$\mid$	$bool$
		$\mid$	$string$
		$\mid$	$date$
Paths	p	$::=$	$.ap \mid /ap \mid \epsilon \mid /$
Query Expressions	q	$::=$	$\mathsf{db}(t, \overline{e})$
		$\mid$	$\mathsf{foreach}_c \{ \overline{x \leftarrow e} \}\ e$
		$\mid$	$\mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \}$
		$\mid$	$\mathsf{do}\ e_{\downarrow p}\{e\}$
		$\mid$	$\mathsf{return}\ e$

Figure 1: Language Syntax

In the remainder of the paper we introduce the language by means of a running example (Section 3), that we then use to illustrate the compilation (Section 7) and simplification process (Section 7.1). We formalize the operational semantics of language λ_{CDL} and its type system in Sections 4 and 5. The localization mechanism is introduced in Section 6, which we prove sound with relation to the language semantics.

Section 7 presents the typed directed compilation and optimization algorithm. Section 7.2 illustrates the code that is actually obtained when querying several different locations, and the glue code that post-processes the results.

2. Syntax

We introduce our common data manipulation language (λ_{CDL}), defined by the syntax given in Figure 1. Its core is a lambda calculus, with records and multisets, equipped with a data manipulation language fragment, capable of querying nested structured data repositories (cf. relational databases, structured JSON data objects, etc.), similar to works using NRC [4, 5]. Our language is based on a set of predefined named data sources t, variables x, y, z, and record labels a, b, c, d. We use the list notation $[\overline{v}]$ to denote the bag construction $[v_1] \uplus [v_2] \ldots \uplus [v_n]$. For the sake of simplicity, conditionals and logic operators are omitted from the syntax, but freely used in the examples.

We introduce a language fragment with expressions that represent queries over parameterized data sources ($\mathsf{db}(t, \overline{e})$). There is a general iteration operation, of the form ($\mathsf{foreach}_c \{ \overline{x \leftarrow e} \}\ e'$), over a set of joined inner queries ($\overline{e}$), with cursors ($\overline{x}$), and filtered by a condition (c). We introduce an operation, of the form ($\mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \}$), that groups the results of an inner query (e) by a set of computed criteria ($\overline{a = e}$) where the label to access the details of each group is also given (b). This operation corresponds to the specification of nested query results, regardless of the underlying support. Cursor x is bound in expressions $\overline{e}$.

In order to manipulate and transform structured nested data we introduce a general purpose operation that operates deep in the nested query results. The operation, of the form ($\mathsf{do}\ e_{\downarrow p}\{e'\}$), applies the abstraction denoted by expression e to the fragments of the resulting values of query e' identified by path p. This operation allows in-place modification of parts of nested results, by iterating or filtering them, joining them with other data-sources, or grouping them with local criteria. We define queries as logically separated values, that can be gradually composed (cf. staged computations [10]) by query operations, and whose base constructor has the form $\mathsf{return}\ e$, and expression $\mathsf{exec}\ x = e\ \mathsf{in}\ e'$ represents the execution of the query denoted by e and binds its results to x in e'. We write $\mathsf{run}\ e$ to abbreviate $\mathsf{exec}\ x = e\ \mathsf{in}\ x$.

3. Example

To illustrate and motivate the language semantics, we use the running example below. Consider a mobile app that organizes the daily tasks of field technicians in a telecom company. It relies on a cloud based system that stores its core data in a relational database, named `SALESDB`, with the sample data sources (relational tables) depicted in Figure 2, and whose schema is as follows:

- `Team` : $\langle id\colon \mathsf{Num}, name\colon \mathsf{String}\rangle^*$
- `Client` : $\langle id\colon \mathsf{Num}, name\colon \mathsf{String}, address\colon \mathsf{String}\rangle^*$
- `Task` : $\langle id\colon \mathsf{Num}, title\colon \mathsf{String}, teamId\colon \mathsf{Num},$ $cliId\colon \mathsf{Num}, date\colon \mathsf{Date}, start\colon \mathsf{Num}, end\colon \mathsf{Num}\rangle^*$

Realtime information about the location of technicians is stored in a MongoDB instance, `TRACKER`, in a collection named `Techs`. A sample record in the database is

```
{ "team": "1", "tech": "Ann", "loc": [51.4, 0.01] }
```

The system also uses a geolocation web service, named `GEO`, to obtain the GPS coordinates for a given street address, which is specified by the following function type:

- `Coords` : $\mathsf{String} \rightarrow \langle lat\colon \mathsf{Num}, lng\colon \mathsf{Num}\rangle$

A developer needs to know the tasks assigned to a team in a given date, e.g., May 8. So, she gradually builds a query. The first step is to join tables `Team` and `Task`, Figure 7a, using a foreach expression, a basic filter, and record constructs.

$$work = \mathsf{foreach}_{e.id=t.teamId \wedge t.date=8/5} \left\{ \begin{array}{l} e \leftarrow teams, \\ t \leftarrow tasks \end{array} \right\} \\ \langle team = e, task = t \rangle$$

Next, the developer groups the results by team's name, with a groupby operation, Figure 7b. The results are a nested collection of records, each containing a team's name, and a list of records (task and team).

$$workByTeam = \mathsf{groupby}_{details}^{name=x.team.name} \{ x \leftarrow work \}$$

The developer adds a new column to the query, yielding the tasks' duration, by means of a in-place modification given the path $/details$. See Figure 8. The in-place operations can easily be defined using a UI, by pointing to the displayed data. A developer using a textual query language would naturally modify the initial query to include the new column.

$$\begin{array}{l} addDuration = \lambda x.\mathsf{foreach}\ \{ y \leftarrow x \} \\ \qquad\qquad (y \oplus \langle dur = y.task.end - y.task.start \rangle) \\ workDur = \mathsf{do}\ addDuration_{\downarrow /details}\{workByTeam\} \end{array}$$

Our approach is better suited to a scenario of a visual manipulation language, or even when the original data is already nested. Moreover, we envisage an automatic simplification process, introduced in section 7.1, that rewrites in-place operations, compacting them as much as possible. The full developments of the simplification are left to future work.

$teams = \mathsf{db}(\texttt{Team})$

id	name
1	Alpha
2	Bravo
3	Charlie

(a) Teams

$clients = \mathsf{db}(\texttt{Client})$

id	name	address
1	Helen	75 Globe Road, London
2	Ive	58 Pitfold Road, London
3	James	4 Dean's Court, London
4	Lewis	25 Ebury Bridge Road, London

(b) Clients

$tasks = \mathsf{db}(\texttt{Task})$

id	title	teamId	cliId	date	start	end
1	Check WiFi	1	2	8/5	10	11
2	Replace phone	1	3	8/5	11	12
3	Setup TV	2	1	8/5	10	11
4	Install router	1	4	8/5	14	16
5	Replace cable	3	4	10/5	9	10

(c) Tasks

Figure 2: Data sources

$$\begin{array}{llll}
\text{Values} & u, v & ::= & \lambda x.e \mid \langle \overline{a = u} \rangle \mid \emptyset \mid [v] \mid v \uplus v \\
 & & \mid & num \mid bool \mid string \mid date \mid r \\
\text{Query values} & r, s & ::= & \mathsf{db}(t, \overline{v}) \mid \mathsf{foreach}_c \{ \overline{x \leftarrow r} \} \ e \\
 & & \mid & \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow r \} \\
 & & \mid & \mathsf{do}\ (\lambda x.e)_{\downarrow p} \{r\} \mid \mathsf{return}\ v
\end{array}$$

Figure 3: Language Values

In our example, we still miss information about the client that each team should visit. So, we modify the current query with an in-place operation using path $/details$. See Figure 9.

$$addClient = \lambda x.\mathsf{foreach}_{y.task.cliId = c.id} \left\{ \begin{array}{l} y \leftarrow x, \\ c \leftarrow clients \end{array} \right\} \\ (y \oplus \langle client = c \rangle)$$

$$withClient = \mathsf{do}\ addClient_{\downarrow /details} \{workDur\}$$

To obtain the GPS coordinate of each client, we call the `Coords` web-service for each one of the addresses (see Figure 10), and modify the query in-place. Finally, using the MongoDB data source that tracks technicians, we obtain the nearest technician to each one of the clients (Figure 11).

Given this last query, we want to dispatch the join between the three tables to the database server running SQL, while the web-service should be called in-memory. Moreover, information about a concrete usage for the data, for instance, not using duration in a given app UI, can be used to discard the expression that computes the duration (which indeed could have impact). In the same way, if it is not important to know the coordinates (e.g. in the back-office), then the call to the web service can be eliminated.

In the following sections we show the semantics of the language, and the corresponding typing relation.

4. Semantics of λ_{CDL}

The operational semantics for λ_{CDL} is defined by a big-step relation on expressions with relation to a state ($\mathcal{S}$), representing referred data repositories. We write $\langle\!\langle e \rangle\!\rangle$ to denote the computed value of an expression e, defined by the grammar in Figure 3, and define it using the cases in Figures 4 and 5. The evaluation of query expressions corresponds to staging queries, that are afterwards executed with relation to the given state, by means of an exec expression. In our scenario, this corresponds to executing queries in remote database systems. We use sets ($\{\overline{e}\}$) and multi-sets ($[\overline{e}]$), with list comprehension notation, as the basis to define the semantics of executing query values r, by the relation $\llbracket r \rrbracket$, defined in Figure 6 (cf. [3–5, 14]).

The call-by-value semantics of most expressions is straightforwardly defined in the structure of the expressions, hence we omit any further explanation. The non-standard construct, $\mathsf{exec}\ x = e\ \mathsf{in}\ e'$, first stages the query value denoted by e, and proceeds with the evaluation of e' binding x to the results of the query (cf. [10]). This is the extension point that we use, later on in this paper, to extend the semantics with the typed compilation procedure, that transforms the queries before actually executing them.

$$\begin{array}{ll}
\langle\!\langle v \rangle\!\rangle = v & \\
\langle\!\langle e\ e' \rangle\!\rangle = \langle\!\langle e''\{^v/_x\} \rangle\!\rangle & \text{where } \langle\!\langle e \rangle\!\rangle = \lambda x.e'' \\
 & \qquad\quad \langle\!\langle e' \rangle\!\rangle = v \\
\langle\!\langle \langle \overline{a = e} \rangle \rangle\!\rangle = \langle \overline{a = \langle\!\langle e \rangle\!\rangle} \rangle & \\
\langle\!\langle e.a \rangle\!\rangle = v & \text{where } \langle\!\langle e \rangle\!\rangle = \langle a = v, ... \rangle \\
\langle\!\langle e \oplus e' \rangle\!\rangle = \langle \overline{a = v}, \overline{b = u} \rangle & \text{where } \langle\!\langle e \rangle\!\rangle = \langle \overline{a = v} \rangle \\
 & \qquad\quad \langle\!\langle e' \rangle\!\rangle = \langle \overline{b = u} \rangle \\
\langle\!\langle [e] \rangle\!\rangle = [\, \langle\!\langle e \rangle\!\rangle \,] & \\
\langle\!\langle e \uplus e' \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle \uplus \langle\!\langle e' \rangle\!\rangle & \\
\langle\!\langle \mathsf{exec}\ x = e\ \mathsf{in}\ e' \rangle\!\rangle = \langle\!\langle e'\{^v/_x\} \rangle\!\rangle & \text{where } r = \langle\!\langle e \rangle\!\rangle \\
 & \qquad\quad v = \llbracket r \rrbracket
\end{array}$$

Figure 4: Operational semantics for expressions

$$\begin{array}{l}
\langle\!\langle \mathsf{db}(t, \overline{e}) \rangle\!\rangle = \mathsf{db}(t, \overline{\langle\!\langle e \rangle\!\rangle}) \\
\langle\!\langle \mathsf{foreach}_c \{ \overline{x \leftarrow e} \} \ e' \rangle\!\rangle = \mathsf{foreach}_c \left\{ \overline{x \leftarrow \langle\!\langle e \rangle\!\rangle} \right\} \ e' \\
\langle\!\langle \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e' \} \rangle\!\rangle = \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow \langle\!\langle e' \rangle\!\rangle \} \\
\langle\!\langle \mathsf{do}\ e_{\downarrow p} \{e'\} \rangle\!\rangle = \mathsf{do}\ e_{\downarrow p} \{ \langle\!\langle e' \rangle\!\rangle \} \\
\langle\!\langle \mathsf{return}\ e \rangle\!\rangle = \mathsf{return}\ \langle\!\langle e \rangle\!\rangle
\end{array}$$

Figure 5: Operational semantics for query expressions

$$\llbracket \mathsf{db}(t, \overline{v}) \rrbracket = (\mathcal{S}(t))(\overline{v})$$

$$\llbracket \mathsf{foreach}_c \{ \overline{x \leftarrow r} \} \ e \rrbracket = [\, \langle\!\langle e\{^{\overline{u}}/_{\overline{x}}\} \rangle\!\rangle \mid \overline{u \in \llbracket r \rrbracket}, \langle\!\langle c\{^{\overline{u}}/_{\overline{x}}\} \rangle\!\rangle \,]$$

$$\begin{array}{l}
\llbracket \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow r \} \rrbracket = [k \oplus \langle b = details_k \rangle \mid k \in keys] \\
\quad \text{where} \\
\quad keys = \{ \langle \overline{a = \langle\!\langle e_a\{^u/_x\} \rangle\!\rangle} \rangle \mid u \in \llbracket r \rrbracket \} \\
\quad details_k = [u \mid u \in \llbracket r \rrbracket, \langle \overline{a = \langle\!\langle e_a\{^u/_x\} \rangle\!\rangle} \rangle = k]
\end{array}$$

$$\llbracket \mathsf{do}\ e_{\downarrow\epsilon} \{r\} \rrbracket = \langle\!\langle e\ (\mathsf{return}\ \llbracket r \rrbracket) \rangle\!\rangle$$

$$\llbracket \mathsf{do}\ e_{\downarrow /} \{r\} \rrbracket = [\, \langle\!\langle e\ (\mathsf{return}\ u) \rangle\!\rangle \mid u \in \llbracket r \rrbracket \,]$$

$$\begin{array}{l}
\llbracket \mathsf{do}\ e_{\downarrow .ap} \{r\} \rrbracket = \langle a = \llbracket \mathsf{do}\ e_{\downarrow p} \{\mathsf{return}\ u\} \rrbracket, \overline{b = v} \rangle \\
\quad \text{where} \\
\quad \llbracket r \rrbracket = \langle a = u, \overline{b = v} \rangle
\end{array}$$

$$\begin{array}{l}
\llbracket \mathsf{do}\ e_{\downarrow /ap} \{r\} \rrbracket = \\
\quad [\, \langle a = \llbracket \mathsf{do}\ e_{\downarrow p} \{\mathsf{return}\ u\} \rrbracket, \overline{b = v} \rangle \mid \langle a = u, \overline{b = v} \rangle \in \llbracket r \rrbracket \,]
\end{array}$$

Figure 6: Operational semantics for query values

Query expressions are interpreted at the top-level as to evaluate their inner expressions that represent queries, producing query values (Figure 5). The semantics of executing query values (Figure 6), states that a data source invocation ($\mathsf{db}(t, \overline{v})$) is represented by directly accessing state $\mathcal{S}$, and calling the data source end point with the given parameters. This general model using sources with

$work = \mathsf{foreach}_{e.id=t.teamId \wedge t.date=8/5} \{ e \leftarrow teams, t \leftarrow tasks \}$
$\langle team = e, task = t\rangle$

team		task				
id	name	id	title	cliId	start	end
1	Alpha	1	Check WiFi	2	10	11
1	Alpha	2	Replace phone	3	11	12
2	Bravo	3	Setup TV	1	10	11
1	Alpha	4	Install router	4	14	16

(a) Teams and tasks for May 8

$workByTeam = \mathsf{groupby}^{name=x.team.name}_{details}\{ x \leftarrow work \}$

name	details					
	team	task				
	...	id	title	cliId	start	end
Alpha	...	1	Check WiFi	2	10	11
	...	2	Replace phone	3	11	12
	...	4	Install router	4	14	16
Bravo	...	3	Setup TV	1	10	11

(b) Group by team's name

Figure 7: Join and group

$workDur = \mathsf{do}\ addDuration_{\downarrow/details}\{workByTeam\}$ where

$addDuration = \lambda x.\mathsf{foreach}\ \{ y \leftarrow x \}\ (y \oplus \langle dur = y.task.end - y.task.start\rangle))$

name	details						
	team	task					dur
	...	id	title	cliId	start	end	
Alpha	...	1	Check WiFi	2	10	11	1
	...	2	Replace phone	3	11	12	1
	...	4	Install router	4	14	16	2
Bravo	...	3	Setup TV	1	10	11	1

Figure 8: Task duration

$withClient = \mathsf{do}\ addClient_{\downarrow/details}\{workDur\}$ where

$addClient = \lambda x.\mathsf{foreach}_{y.task.cliId=c.id}\ \{ y \leftarrow x, c \leftarrow clients \}\ (y \oplus \langle client = c\rangle))$

name	details					
	team	task		dur	client	
	...	cliId	...		name	address
Alpha	...	2	...	1	Ive	58 Pitfold Road, London
	...	3	...	1	James	4 Dean's Court, London
	...	4	...	2	Lewis	25 Ebury Bridge Road, London
Bravo	...	1	...	1	Helen	75 Globe Road, London

Figure 9: Join with Clients

$withLoc = \mathsf{do}\ addLoc_{\downarrow/details}\{withClient\}$ where

$addLoc = \lambda x.\mathsf{foreach}\ \{ y \leftarrow x \}\ (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}(\mathtt{Coords}, y.client.address)\rangle))$

name	details							
	team	task		dur	client		loc	
	...	cliId	...		name	address	lat	lng
Alpha	...	2	...	1	Ive	58 Pitfold Road, London	51.45	0.02
	...	3	...	1	James	4 Dean's Court, London	51.52	-0.15
	...	4	...	2	Lewis	25 Ebury Bridge Road, London	51.50	-0.15
Bravo	...	1	...	1	Helen	75 Globe Road, London	51.52	-0.05

Figure 10: Get address coordinates

parameters allows the representation of both web-services requiring parameters, and database tables which do not. The execution of an iteration includes joining the results of inner queries, producing and filtering a value for each tuple. Group operations compute the unique values given by the grouping criteria (i.e. the keys), and use them to produce a nested structure, which pairs each key with a details field containing all the original values that are grouped under it.

The semantics of operations of the form $\mathsf{do}\ e_{\downarrow p}\{r\}$, is defined by case analysis of the path given. In the case of the empty path, it is mapped onto applying the abstraction denoted by expression e to the results of query e'. The case where the path is $/$, corresponds to a map operation, applying the abstraction for each of the elements in the collection. The remaining cases of $.ap$ and $/ap$ navigate in the structure of the target value, and apply the operation.

5. Typing

We define the type language for λ_{CDL} as follows,

$$\begin{array}{rcl} \tau, \sigma & ::= & \mathsf{Num} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Date} \mid \langle \overline{a : \tau} \rangle \mid \tau^* \mid \tau \rightarrow \sigma \\ & \mid & \mathcal{Q}(\tau) \end{array}$$

and define a typing relation, expressed by the judgment $\Delta \vdash e : \tau$, and defined by the rules in Figure 12. We use basic types for integer numbers, strings, and dates, to match our running example. We follow standard lines to type abstractions, records, and multisets, and therefore omit these rules for the sake of saving space, please refer to [15] to find a complete definition. The notable feature of the type system is that query expressions returning a value of type τ are typed with a special type $\mathcal{Q}(\tau)$, whose resulting data can be obtained by expression exec, rule (EXEC).

$withTech = \mathsf{do}\ addTech_{\downarrow/details}\{withLoc\}$ where

$addTech = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ (y \oplus \langle tech = (\mathsf{run}\ getTech(y.team.id, y.loc))[0]\rangle)$

$getTech = \lambda t \lambda l.\mathsf{foreach}_{x.teamId=t \wedge near(l.lat,l.lng,x.loc[0],x.loc[1],10)}\ \{\ x \leftarrow \mathsf{db}(Techs)\}\ x$

name	details											
	team		task	dur	client		loc		tech		
	...	id	...		name	...	lat	lng	team	tech	loc
Alpha	...	1	...	1	Ive	...	51.45	0.02	1	Ann	[51.4, 0.01]
	...	1	...	1	James	...	51.52	-0.15	1	Bob	[51.5, -0.1]
	...	1	...	2	Lewis	...	51.50	-0.15	1	Bob	[51.5, -0.1]
Bravo	...	2	...	1	Helen	...	51.52	-0.05	2	David	[51.5, -0.03]

Figure 11: Get technician information near client information

Rule (AT) types an operation that is applied, in-place, deep in the structure of a query. The following definition, follows the structure of the type, matches the type at the end of a path, and applies the given type transformation. The operation applies a query transformation operation, of type $\mathcal{Q}(\tau') \to \mathcal{Q}(\sigma')$, and the operation on types $\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$, validates and transforms the necessary "deep" transformation of the target query.

Definition 1 (Type at)**.**

$$\begin{array}{rcl} \tau_{\downarrow\epsilon}\{^{\tau}/_{\sigma}\} & = & \sigma \\ \tau^*{}_{\downarrow/}\{^{\tau}/_{\sigma}\} & = & \sigma^* \\ (\langle a:\tau\rangle \oplus \sigma)_{\downarrow.ap}\{^{\tau'}/_{\sigma'}\} & = & (\langle a:\tau_{\downarrow p}\{^{\tau'}/_{\sigma'}\}\rangle \oplus \sigma) \\ (\langle a:\tau\rangle \oplus \sigma)^*_{\downarrow/ap}\{^{\tau'}/_{\sigma'}\} & = & (\langle a:\tau_{\downarrow p}\{^{\tau'}/_{\sigma'}\}\rangle \oplus \sigma)^* \end{array}$$

This typing relation is sound with relation to the language semantics as expressed by the standard Theorem 1.

Theorem 1 (Type preservation)**.**

1. *If* $\Delta \vdash e : \tau$ *and* $\langle\!\langle e \rangle\!\rangle = v$ *then* $\Delta \vdash v : \tau$*.*
2. *If* $\Delta \vdash r : \mathcal{Q}(\tau)$ *and* $[\![r]\!] = v$ *then* $\Delta \vdash v : \tau$*.*

This result is proven by the usual induction strategy on the typing and type transformation definitions, and supports the usual properties of absence of runtime errors for terminating expressions. Proofs and intermediate lemmas are available in the companion technical report [15].

6. Localization

Optimizations are a well-known problem in relational databases, with many variants [21] that shape the execution plan in order to optimize the usage of memory and CPU time. In a distributed and heterogeneous setting, the criteria to optimize a query's execution plan are somewhat different. The way different data sources are interplayed can shorten the execution time of a query in a significant way because the determining factor is no longer memory usage and CPU time, but the amount of data that is interchanged through the network, the number of locations visited, and the native capabilities used on each database system or data repository.

We next extend the data manipulation language introduced in section 2 with a location and type based transformation process for queries. Queries are transformed in such a way that subexpressions are grouped to be shipped to remote locations, and executed in the most efficient way possible. We use knowledge about the capabilities of each remote site [18], in order to place the operations as close as possible to the origin of the data. The parts of a query that can be computed remotely are grouped and dispatched, and an *in-memory* post-processing phase is generated to complete the job, in the starter location. We leverage not only on the locations of data sources, but also on the actual usage of data, which is expressed as type information. The transformation process prunes the query tree, to avoid fetching unnecessary data, and eliminates all remote invocations that have impact on the processing time but not in the output data. We divide the compilation process into the eager localization of the query components, and in the use of type information to prune parts of the query. For an optimized distributed execution we foresee that we can use orthogonal strategies to efficiently execute it (e.g. [12]).

6.1 Typed Localization

Consider a global mapping Γ from data source names to a set of locations $\ell \in \mathcal{L}$, and assume that there is a location $\top$ that represents the starting location, where all computations are explicitly performed in memory. We consider also a set of predefined predicates to specify capabilities of locations. The truth value of the predicates is predetermined and immutable. The selection of predicates used here is inspired on the concrete experience of developing a DSL [17] for data manipulation, and is adapted to the set of operations that is included in the language. We say that proposition `can_group`(ℓ) holds if the database engine running at location ℓ is able to execute a groupby operation with aggregation of results, as in relational databases. Predicate `can_nestgroups`(ℓ) holds for locations (ℓ) running database engines which support for the nested grouping operations, i.e. return a query together with the details of its groups. This is the case of some NoSQL databases such as MongoDB. Predicate `can_join`(ℓ) states that the database repository at location ℓ supports the joining of two (or more) sources given a condition, and `can_iterate`(ℓ) indicates that it supports the iteration of a list and the computing of a given expression on all elements of a query. As an example, consider a classic REST interface, yielding a JSON object. None of the above predicates holds since the interface's only capability is to return the data.

To express such localization relation, we define a type directed relation that states that an expression e can be remotely evaluated at location ℓ, to produce data of type τ, with relation to a location environment Γ, and a typing environment Δ, which is written as follows

$$\Delta, \Gamma \vdash e : \tau \rightsquigarrow \ell$$

and is defined by the rules in Figure 13. Initial typing and location environments, Δ_0 and Γ_0, are set as to contain references to predefined and localized data sources. Predefined localized functions can also be assumed to exist in the typing and localizing environments. For instance, SQL databases provide function `NOW()` and MongoDB provides specialized operators such as `$near` to compare GPS coordinates.

Notice in rule (L-ID) that all well typed and localized identifiers are both assigned a type (in Δ) and a location (in Γ). Syntax forces data-source identifiers (t) to be separated from variables (x), rule (L-ID) is not applicable to data sources. We assume that numbers can be trivially used in all query languages across all locations, rule (L-NUM). The definition of anonymous functions is dependent on the location and its host query language (premise `can_lambda`(ℓ) in rule (L-FUN)). As an example, anonymous func-

$$(\textsc{Source})\ \frac{\Delta \vdash e_i : \tau_i \quad {\scriptstyle i=1..n}}{\Delta, t : \overline{\tau} \to \tau \vdash \mathsf{db}(t, \overline{e}) : \mathcal{Q}(\tau)} \qquad (\textsc{Group})\ \frac{\Delta \vdash e : \mathcal{Q}(\tau^*) \quad \Delta, x : \tau \vdash e_i : \sigma_i \quad {\scriptstyle i=1..n}}{\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\, x \leftarrow e \,\} : \mathcal{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)}$$

$$(\textsc{Select})\ \frac{\Delta \vdash e_i : \mathcal{Q}(\tau_i^*)\ {\scriptstyle i=1..n} \quad \Delta, \overline{x : \tau} \vdash e' : \mathsf{Bool} \quad \Delta, \overline{x : \tau} \vdash e'' : \sigma}{\Delta \vdash \mathsf{foreach}_{e'}\ \{\, \overline{x \leftarrow e} \,\}\ e'' : \mathcal{Q}(\sigma^*)} \qquad (\textsc{Return})\ \frac{\Delta \vdash e : \tau}{\Delta \vdash \mathsf{return}\ e : \mathcal{Q}(\tau)}$$

$$(\textsc{At})\ \frac{\Delta \vdash e : \mathcal{Q}(\tau') \to \mathcal{Q}(\sigma') \quad \Delta \vdash e' : \mathcal{Q}(\tau)}{\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathcal{Q}(\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\})} \qquad (\textsc{Exec})\ \frac{\Delta \vdash e : \mathcal{Q}(\sigma) \quad \Delta, x : \sigma \vdash e' : \tau}{\Delta \vdash \mathsf{exec}\ x = e\ \mathsf{in}\ e' : \tau}$$

Figure 12: Typing relation (partial)

$$(\textsc{L-Mem})\ \frac{\Delta \vdash e : \tau}{\Delta; \Gamma \vdash e : \tau \rightsquigarrow \top} \qquad (\textsc{L-Num})\ \Delta; \Gamma \vdash num : \mathsf{Num} \rightsquigarrow \ell \qquad (\textsc{L-Id})\ \Delta, x : \tau; \Gamma, x : \ell \vdash x : \tau \rightsquigarrow \ell$$

$$(\textsc{L-Fun})\ \frac{\Delta, x : \tau; \Gamma, x : \ell \vdash e : \sigma \rightsquigarrow \ell \quad \mathtt{can\_lambda}(\ell)}{\Delta; \Gamma \vdash \lambda x.e : \tau \to \sigma \rightsquigarrow \ell} \qquad (\textsc{L-App})\ \frac{\Delta; \Gamma \vdash e' : \tau \rightsquigarrow \ell \quad \Delta; \Gamma \vdash e : \tau \to \sigma \rightsquigarrow \ell \quad \mathtt{can\_call}(\ell)}{\Delta; \Gamma \vdash e\ e' : \sigma \rightsquigarrow \ell}$$

$$(\textsc{L-Record})\ \frac{\Delta; \Gamma \vdash e_i : \tau_i \rightsquigarrow \ell \ \ {\scriptstyle i=1..n}}{\Delta; \Gamma \vdash \langle \overline{a = e} \rangle : \langle \overline{a : \tau} \rangle \rightsquigarrow \ell} \qquad (\textsc{L-Field})\ \frac{\Delta; \Gamma \vdash e : \langle a : \tau, \overline{b : \sigma} \rangle \rightsquigarrow \ell}{\Delta; \Gamma \vdash e.a : \tau \rightsquigarrow \ell}$$

$$(\textsc{L-Concat})\ \frac{\Delta; \Gamma \vdash e : \langle \overline{a : \tau} \rangle \rightsquigarrow \ell \quad \Delta; \Gamma \vdash e' : \langle \overline{b : \sigma} \rangle \rightsquigarrow \ell}{\Delta; \Gamma \vdash e \oplus e' : \langle \overline{a : \tau}, \overline{b : \sigma} \rangle \rightsquigarrow \ell}\ \overline{a} \# \overline{b} \qquad (\textsc{L-Empty})\ \frac{\mathtt{can\_createlists}(\ell)}{\Delta; \Gamma \vdash \emptyset : \tau^* \rightsquigarrow \ell}$$

$$(\textsc{L-Singleton})\ \frac{\Delta; \Gamma \vdash e : \tau \rightsquigarrow \ell \quad \mathtt{can\_createlists}(\ell)}{\Delta; \Gamma \vdash [e] : \tau^* \rightsquigarrow \ell} \ (\textsc{Append})\ \frac{\Delta; \Gamma \vdash e : \tau^* \rightsquigarrow \ell \quad \Delta; \Gamma \vdash e' : \tau^* \rightsquigarrow \ell \quad \mathtt{can\_createlists}(\ell)}{\Delta; \Gamma \vdash e \uplus e : \tau^* \rightsquigarrow \ell}$$

$$(\textsc{L-Source})\ \frac{\Delta; \Gamma \vdash e_i : \tau_i \rightsquigarrow \ell \ \ {\scriptstyle i=1..n}}{\Delta, t : \overline{\tau} \to \tau; \Gamma, t : \ell \vdash \mathsf{db}(t, \overline{e}) : \mathcal{Q}(\tau) \rightsquigarrow \ell} \qquad (\textsc{Return})\ \frac{\Delta; \Gamma \vdash e : \tau \rightsquigarrow \ell}{\Delta; \Gamma \vdash \mathsf{return}\ e : \mathcal{Q}(\tau) \rightsquigarrow \ell}$$

$$(\textsc{L-Group})\ \frac{\Delta; \Gamma \vdash e : \mathcal{Q}(\tau^*) \rightsquigarrow \ell \quad \Delta, x : \tau; \Gamma, x : \ell \vdash e_i : \sigma_i \rightsquigarrow \ell \ \ {\scriptstyle i=1..n} \quad \mathtt{can\_group}(\ell)}{\Delta; \Gamma \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\, x \leftarrow e \,\} : \mathcal{Q}(\langle \overline{a : \sigma} \rangle^*) \rightsquigarrow \ell}$$

$$(\textsc{L-Details})\ \frac{\Delta; \Gamma \vdash e : \mathcal{Q}(\tau^*) \rightsquigarrow \ell \quad \Delta, x : \tau; \Gamma, x : \ell \vdash e_i : \sigma_i \rightsquigarrow \ell \ \ {\scriptstyle i=1..n} \quad \mathtt{can\_nestgroups}(\ell)}{\Delta; \Gamma \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\, x \leftarrow e \,\} : \mathcal{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*) \rightsquigarrow \ell}$$

$$(\textsc{L-Select})\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e_i : \mathcal{Q}(\tau_i^*) \rightsquigarrow \ell \ \ {\scriptstyle i=1..n} \quad (\mathtt{can\_join}(\ell) \lor n = 1) \\ \Delta, \overline{x : \tau}; \Gamma, \overline{x : \ell} \vdash e' : \mathsf{Bool} \rightsquigarrow \ell \quad \Delta, \overline{x : \tau}; \Gamma, \overline{x : \ell} \vdash e'' : \sigma \rightsquigarrow \ell \quad \mathtt{can\_iterate}(\ell)\end{array}}{\Delta; \Gamma \vdash \mathsf{foreach}_{e'}\ \{\, \overline{x \leftarrow e} \,\}\ e'' : \mathcal{Q}(\sigma^*) \rightsquigarrow \ell}$$

Figure 13: Type directed localization

tions are supported in locations running MongoDB with Javascript, or even in (imaginary) locations accepting LINQ queries containing $C\sharp$ lambda expressions, but, is not accepted in locations based on SQL. Calling functions ($e\ e'$) is also location dependent, that holds for SQL locations if the called function (e) refers to a predefined SQL function of the appropriate type.

Besides the predefinition of localized constants, the support for localizing expressions is axiom (L-Source), which assigns a location and a type to a data source. Rule (L-Group) asserts that a groupby operation can be computed at a certain location ℓ, depending on its sub-expressions and the capability of grouping of the given location. Rule (L-Group) is focused on the so-called top-level attributes of a groupby operation (the grouping criteria), if the intended type refers to the details of the groups, then rule (L-Details) requires a different capability from the location. Notice type $\mathcal{Q}(\langle \overline{a : \sigma}, b\ :\ \tau^* \rangle)$, referring to label b, and the different predicate $\mathtt{can\_nestgroups}(\ell)$.

Not all database source locations are capable of iterating and filtering data sources. For instance, a web-service API, may not include services for filtering or iterating its provided data. Rule (L-Select) localizes expressions based on iteration and filtering capabilities (predicate $\mathtt{can\_iterate}(\ell)$). Locations with iteration capabilities may, nonetheless, lack the ability of joining two independent sources (e.g. MongoDB if the used adapter does not allow it, or indexedDB if special indexes are not used), which is reflected in rule (L-Select)'s premise $\mathtt{can\_join}(\ell)$. Note also that subqueries must be located at the same location as the select and condition expressions. So far, we limit the delegation of subqueries to arbitrary locations only to the $\top$ location (through rule (L-Mem)). A lattice of locations, with a delegation relation, can be used to establish a more flexible localization relation, which would model database engines supporting mechanisms similar to linked servers [2]. Capability $\mathtt{can\_join}(\ell)$ is only considered when more than one source is used.

Notice that we assume that all well-typed expressions may be computed in memory, as expressed in rule (L-Mem). This means that, the localizing relation compositionally assigns a location to each subexpression of a query and falls-back to assigning the memory location ($\top$) when it is no longer possible to locate a query in a single place, either because data-sources from different locations are used, or because some capability is simply not available at a given location. As shown in section 3, a query may be transformed to localize groupby operations directly into one target database engine, or it may be forced to collect all remote data and explicitly group it locally (in the application server or client application). This may happen when referring to details data in a relational database, using a given function to filter that is not known in the target location, or when the location does not have grouping capabilities with the given criteria.

To the best of our knowledge there are no database engines that perform in-place operations as defined in λ_{CDL}. Thus, in-place operations are localized at $\top$ (in-memory).

We also consider the structural subtyping relation

$$\frac{\tau_i \leq \tau_i' \quad {\scriptstyle i=1..n}}{\langle \overline{a:\tau}, \overline{b:\sigma}\rangle \leq \langle \overline{a:\tau'}\rangle} \qquad \frac{\tau \leq \tau'}{\mathcal{Q}(\tau) \leq \mathcal{Q}(\tau')}$$

$$\frac{\tau' \leq \tau \quad \sigma \leq \sigma'}{\tau \to \sigma \leq \tau' \to \sigma'} \qquad \frac{\tau \leq \sigma}{\tau^* \leq \sigma^*}$$

which we use in the following soundness lemma.

Lemma 2 (Soundness of Localization)**.**
$\Delta, \Gamma \vdash e : \tau \rightsquigarrow \ell$ *iff* $\Delta \vdash e : \sigma$ *and* $\sigma \leq \tau$.

This lemma holds trivially in all cases but rule (L-DETAILS). In this case the type is coerced to not contain the details, which is obviously a supertype. We prove the "if" part of this lemma by induction on the size of the type derivations and by analysis of the last case used. Notice that subtyping is not introduced initially in the language as an universal law, instead we are introducing explicit type coercions (projections) in the query transformation process that follows. The "only if" part of the lemma is proven by the fact that all expressions can be localized in memory ($\top$).

7. Location Based Compilation

We now define a query transformation algorithm that identifies where each part of a query should be executed, guided by a localization mapping of data-sources. The algorithm identifies and isolates parts of a query that should be remotely executed, it separates the code that aggregates and prepares the query results (if possible), or generates new glue code to be executed in the starter location (if necessary).

We define a typed localization relation for all kinds of expressions with relation to the capabilities of locations to execute them (e.g. a SQL location cannot execute function `YearOfDate()`, and therefore a filter using that kind of function should be made in memory). The compilation algorithm, written $(\!|\ r\ |\!)^{\tau}_{\ell}$, is defined on query values, with relation to a type τ, that represents the actual usage of the query results, and a starter location ℓ. It yields a localized expression, where its sub-expressions are tagged to be remotely executed whenever possible, and transformed to execute locally if needed. To represent the output of the algorithm, we extend the language with a new expression $[\ e\ ]_\ell$ whose semantics is to execute expression e at location ℓ. Formally, the semantics of a localized expression is the same as the enclosed expression:

$$\langle\!\langle [\ e\ ]_\ell \rangle\!\rangle \triangleq \langle\!\langle e \rangle\!\rangle$$

We also introduce a projection operation ($\pi^{\tau}_{\sigma}(e)$) that coerces the value of expression e from type σ to type τ. The type based projection is defined on expressions, and either recursively changes the denoted value of the expression, or rewrites the record construction expressions to contain less labeled fields. Proper placement of projection operations can largely improve the efficiency of a query, by

$$(\!|\ x \leftarrow e, B\ |\!)^{\delta,\overline{\delta}}_{\ell} \triangleq \begin{cases} [\ x \leftarrow (\!|\ e\ |\!)^{\delta}_{\ell'}, \overline{x \leftarrow e}\ ]_{\ell'}, B' \\ \qquad\qquad \text{if } \texttt{can\_join}(\ell') \\ \\ x \leftarrow (\!|\ e\ |\!)^{\tau}_{\ell}, [\ \overline{x \leftarrow e}\ ]_{\ell'}, B' \\ \qquad\qquad \text{otherwise} \end{cases}$$

where $(\!|\ B\ |\!)^{\overline{\delta}}_{\ell} = [\ \overline{x \leftarrow e}\ ]_{\ell'}, B'$
and $\Gamma \vdash e : \tau \rightsquigarrow \ell'$

Figure 14: Binder compilation

$$\begin{array}{rl}
{}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(e) = & \text{if } \ell = \ell' \text{ then} \\
& \quad \text{if } \tau = \sigma \text{ then } e \\
& \quad \text{else if } \tau \leq \sigma \text{ then} \\
& \quad\quad \text{if } \texttt{can_project}(\ell) \text{ then } \pi^{\tau}_{\sigma}(e) \\
& \quad\quad \text{else undefined} \\
& \quad \text{else undefined} \\
& \text{else if } \tau = \sigma \text{ then } [\ e\ ]_\ell \\
& \text{else if } \tau \leq \sigma \text{ then} \\
& \quad \text{if } \texttt{can_project}(\ell') \text{ then } [\ \pi^{\tau}_{\sigma}(e)\ ]_{\ell'} \\
& \quad \text{else if } \texttt{can_project}(\ell) \text{ then } \pi^{\tau}_{\sigma}([\ e\ ]_\ell) \\
& \quad \text{else undefined} \\
& \text{else undefined}
\end{array}$$

Figure 15: Projection and localization function

pruning several fields, and avoiding the remote invocation of subqueries. The full definition, in [15], of $\pi^{\tau}_{\sigma}(e)$ is straightforward.

In order to define the compilation algorithm, we introduce in Figure 15 an auxiliary projection and localization function on expressions to define a localized projection of a query expression. We write ${}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(e)$ to denote a projection from type σ to type τ, and from an inner location ℓ' to an outer location ℓ. This operation is always defined for well-typed and localized expressions, where $\sigma \leq \tau$ and $\texttt{can\_project}(\ell)$. The compilation algorithm, in Figure 16, is inductively defined on the structure of a query expression, satisfying the precondition above. It ensures that, either the starting location can apply projections, or the expected type does not require any projection. We express the soundness of the projection and localization function as follows

Lemma 3 (Soundness of projection)**.**
${}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(e)$ *is defined if* $\sigma \leq \tau$ *and* $\texttt{can\_project}(\ell)$.

In all cases, of Figure 16, the resulting projection is directed to the location given by the typed localization relation (ℓ'), and the inner queries are also compiled with the target location ℓ' as starting point. The required usage type for the inner expression depends on the capability of the target location to make projections. Notice that the typing relation can be used to determine the minimum usage type, the greatest supertype of all partial usages of the free variables of an expression. In practice, this corresponds to a simple inference typing algorithm, that starts with an expected type and assigns the usage types to the intermediate steps. In the case of a foreach expression, we use the minimum usage type of each cursor variable, to compile each inner-query, thus either the inner query is compiled with a projection, or the projection is placed together with the foreach expression. In the case of a groupby expression, we use the minimum usage of the cursor in the group criteria expressions in a similar way. In summary, the invariant of the algorithm leads to correct placement of projections, possibly several query layers above the source and usages of the data.

In the case of a foreach expression, the list of inner queries (binders) of the query is compiled according to the definition of

$$(\!|\ \mathsf{db}(t,\overline{v})\ |\!)^{\tau}_{\ell} \triangleq {}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(\mathsf{db}(t,\overline{v})) \quad \text{where} \quad \Delta_0;\Gamma \vdash \mathsf{db}(t,\overline{v}) : \mathcal{Q}(\sigma) \rightsquigarrow \ell' \text{ with } \Gamma(t) = \ell'$$

$$(\!|\ \mathsf{foreach}_c\,\{\ \overline{x \leftarrow r}\ \}\ e\ |\!)^{\tau^*}_{\ell} \triangleq {}^{\tau^*}_{\ell}\Pi^{\sigma^*}_{\ell'}(\mathsf{foreach}_c \left\{\ (\!|\ \overline{x \leftarrow r}\ |\!)^{\overline{\delta''}}_{\ell'}\ \right\}\ e) \quad \text{where}$$
$$\begin{array}{l}
\Delta_0;\Gamma \vdash \mathsf{foreach}_c\,\{\ \overline{x \leftarrow r}\ \}\ e : \mathcal{Q}(\sigma^*) \rightsquigarrow \ell' \\
\forall_{x\in\overline{x}}\ \Delta_0;\Gamma \vdash r_x : \delta'_x \rightsquigarrow \ell' \\
\Delta_0, \overline{x:\delta} \vdash e : \tau \\
\Delta_0, \overline{x:\delta} \vdash c : \mathsf{bool} \quad \text{where } \delta_x \text{ is the minimum usage of each cursor } x \text{ to type expression } e \text{ with } \tau \\
\forall_{x\in\overline{x}} \text{ if } \mathtt{can_project}(\ell') \text{ then } \delta''_x = \delta_x \text{ else } \delta''_x = \delta'_x
\end{array}$$

$$(\!|\ \mathsf{groupby}^{\overline{a=e}}_{b}\{\ x \leftarrow r\ \}\ |\!)^{\tau}_{\ell} \triangleq {}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(\mathsf{groupby}^{\overline{a=e}}_{b}\{\ x \leftarrow (\!|\ r\ |\!)^{\delta''}_{\ell'}\ \}) \quad \text{where}$$
$$\begin{array}{l}
\Delta_0;\Gamma \vdash \mathsf{groupby}^{\overline{a=e}}_{b}\{\ x \leftarrow r\ \} : \mathcal{Q}(\sigma) \rightsquigarrow \ell' \\
\forall_{a\in\overline{a}}\ \Delta_0, x:\delta' \vdash e_a : \tau_a \rightsquigarrow \ell' \\
\forall_{a\in\overline{a}}\ \Delta_0, x:\delta_a \vdash e_a : \tau_a \quad \text{where } \delta_a \text{ represents the minimum usage of cursor } x \text{ to type each expression } e_a \text{ with } \tau_a \\
\exists_{\delta}\forall_{a\in\overline{a}}\ \delta \leq \delta_a \text{ is the meet of all types } \delta_a \text{ in the subtyping relation} \\
\text{if } \mathtt{can_project}(\ell') \text{ then } \delta'' = \delta \text{ else } \delta'' = \delta'
\end{array}$$

$$(\!|\ \mathsf{do}\ e_{\downarrow p}\{r\}\ |\!)^{\tau}_{\ell} \triangleq {}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(\mathsf{do}\ e_{\downarrow p}\{(\!|\ r\ |\!)^{\sigma'}_{\ell'}\}) \quad \text{where} \quad \begin{array}{l}
\Delta_0;\Gamma \vdash \mathsf{do}\ e_{\downarrow p}\{r\} : \mathcal{Q}(\sigma) \rightsquigarrow \ell' \\
\Delta \vdash e : \mathcal{Q}(\delta) \rightarrow \mathcal{Q}(\delta') \\
\exists_{\sigma'}\ \sigma' = \tau_{\downarrow p}\{^{\delta'}/_{\delta}\}
\end{array}$$

$$(\!|\ \mathsf{return}\ v\ |\!)^{\tau}_{\ell} \triangleq {}^{\tau}_{\ell}\Pi^{\sigma}_{\ell'}(\mathsf{return}\ v) \quad \text{where} \quad \Delta_0;\Gamma \vdash \mathsf{return}\ v : \mathcal{Q}(\sigma) \rightsquigarrow \ell'$$

Figure 16: Compilation and localization algorithm

Figure 14. We use meta-variables B and B' to represent sets of binders. The algorithm works by compiling sets of commonly localized binders, depending on the capability of the location to join several data sources. The compilation of the binders of a foreach expression, as in Figure 16:

$$\mathsf{foreach}_c \left\{\ (\!|\ \overline{x \leftarrow r}\ |\!)^{\overline{\delta''}}_{\ell'}\ \right\}\ e$$

and given that the compilation of the binders has the form

$$(\!|\ \overline{x \leftarrow r}\ |\!)^{\overline{\delta''}}_{\ell'} = \overline{x' \leftarrow r'}, [\ \overline{x'' \leftarrow r''}\ ]_{\ell}, \ldots$$

the expanded version of the foreach expression is as follows

$$\mathsf{foreach}_c \left\{ \overline{x' \leftarrow r'}, z \leftarrow [\ \mathsf{foreach}\ \left\{\ \overline{x'' \leftarrow r''}\ \right\}]_{\ell}, \ldots \right\} \\ (e\{^{x_1}/_{z.x_1}\}\{^{\cdots}/_{\ldots}\})$$

The different binders are grouped according to their possible locations, and some are not localized at all.

One aspect that we left out of this paper, for the sake of simplicity, but has a high potential impact, is the separation of the conditions among the several binders of a foreach expression. The compilation strategy is similar to the compilation of query binders, based on the free names of each expression, where an expression

$$\mathsf{foreach}_{c_1 \wedge c_2 \wedge \ldots}\ \{\ x_1 \leftarrow e_1, x_2 \leftarrow e_2, \ldots\ \}\ e$$

can be transformed into

$$\begin{array}{l}\mathsf{foreach}_c\{\ x_1 \leftarrow [\ \mathsf{foreach}_{c_1}\{\ y \leftarrow e_1\ \}\ y\ ]_{\ell}, \\ \quad\quad z\ \leftarrow \mathsf{foreach}_{c_2}\ \{\ x_2 \leftarrow e_2, \ldots\ \}\ \ldots,\ \ldots\}\ e\end{array}$$

The compilation algorithm is designed to interpolate the execution of queries in an extended language semantics, where a usage type, introduced here as a type annotation, is used to guide it

$$\langle\!\langle \mathsf{exec}\ x : \tau = e\ \mathsf{in}\ e' \rangle\!\rangle = \langle\!\langle e'\{^{v}/_{x}\} \rangle\!\rangle \quad \text{where } \begin{array}{l} r = \langle\!\langle e \rangle\!\rangle \\ v = [\![(\!|\ r\ |\!)^{\tau}_{\top}]\!] \end{array}$$

We enunciate the soundness of the definition above as follows

Theorem 4 (Soundness of compilation process)**.**
If $\Delta_0 \vdash r : \mathcal{Q}(\sigma)$ *then* $[\![(\!|\ r\ |\!)^{\tau}_{\top}]\!] = [\![\pi^{\tau}_{\sigma}(r)]\!]$ *with* $\sigma \leq \tau$.

We prove this by case analysis of the compilation function [15]. In summary, the compilation algorithm that we present above produces the query in Figure 17, when applied to the query of our running example in Figure 11.

7.1 Query Simplification

Our running example (Figures 2 to 11) follows a particular sequence of steps, as it mimics the actions of a developer using an interactive query construction tool. Notice that the resulting query (Figure 17) can be simplified to produce another query that's simpler, but equivalent (Figure 18). In this case, the new attribute dur was added after the application of operation groupby, using path $/details$. An equivalent query can be written with the computed attribute being expressed together with the join of `Team` and `Task` data sources. The addition of attribute $client$, by joining data source `Client`, can also be computed together with the first join. Adding the attribute loc, however, cannot be simplified, as it requires calling a web-service, a capability that relational databases do not have. Also, joining technician data involves querying a remote location, which cannot be performed by a relational database, and hence is not included in the inner query.

7.2 Code Generation

When compiling a query we take advantage of the way its results are being used. As an example, consider the query $withLoc$ from Figure 11, that when is localized using the complete type as possible usage, results in the query in Figure 17. If we compile it using $\tau = \langle name : String\rangle^*$ as usage target, then the calls to the `GEO` and `TRACKER` locations can be safely ignored, resulting in the (abbreviated) query:

$$(\!|\ \mathsf{do}\ addTech_{\downarrow/details}\{\ldots\}\ |\!)^{\langle name:String\rangle^*}_{\top} = \\ \left[\mathsf{groupby}^{name=x.team.name}_{details}\{\ldots\}\right]_{\mathtt{SALESDB}}$$

Notice that all in-place operations are eliminated by checking if the given path exists in the target type, and that the groupby operation is compiled and localized in the `SALESDB` database, since the usage does not refer to group details. This query can then be used to produce the C# code shown in Figure 19. If we instead compile it with relation to type

$$\Big[\mathsf{do}\ addTech_{\downarrow/details}\{\ \mathsf{do}\ addLoc_{\downarrow/details}\{\mathsf{do}\ addClient_{\downarrow/details}\{\mathsf{do}\ addDuration_{\downarrow/details}\{$$
$$\mathsf{groupby}^{name=x.team.name}_{details}\{x \leftarrow \Big[\mathsf{foreach}_{e.id=t.teamId \wedge t.date=8/May}\ \{\ e \leftarrow \mathsf{db}(\mathtt{Team}), t \leftarrow \mathsf{db}(\mathtt{Task})\ \}\ \langle team = e, task = t\rangle\Big]_{\mathtt{SALESDB}}\ \}\}\}\}\Big]_{\top}$$

where

$$addTech = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ \ (y \oplus \langle tech = (\mathsf{run}\ [\ getTech(y.team.id, y.loc)\ ]_{\mathtt{TRACKER}})[0]\rangle)$$
$$getTech = \lambda t\lambda l.[\ \mathsf{foreach}_{x.teamId=t \wedge near(l.lat,l.lng,x.loc[0],x.loc[1],10)}\ \{\ x \leftarrow \mathsf{db}(Techs)\}\ \ x\ ]_{\mathtt{TRACKER}}$$
$$addLoc = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ \ (y \oplus \langle loc = \mathsf{run}\ [\ \mathsf{db}(\mathtt{Coords}, y.client.address)\ ]_{\mathtt{GEO}}\rangle)$$
$$addClient = \lambda x.\mathsf{foreach}_{y.task.cliId=c.id}\ \{\ y \leftarrow x, c \leftarrow [\ \mathsf{db}(\mathtt{Client})\ ]_{\mathtt{SALESDB}}\ \}\ \ (y \oplus \langle client = c\rangle)$$
$$addDuration = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ \ (y \oplus \langle dur = y.task.end - y.task.start\rangle)$$

Figure 17: Full localized query

$$\mathsf{do}\ addTech_{\downarrow/details}\{\mathsf{do}\ addLoc_{\downarrow/details}\{\mathsf{groupby}^{name=x.team.name}_{details}\{\ x \leftarrow$$
$$\mathsf{foreach}_{e.id=t.teamId \wedge t.date=8/May \wedge t.cliId=c.id}\ \{\ e \leftarrow \mathsf{db}(\mathtt{Team}), t \leftarrow \mathsf{db}(\mathtt{Task}), c \leftarrow \mathsf{db}(\mathtt{Client})\}$$
$$\langle team = e, task = t, client = c, dur = t.end - t.start\rangle\ \}\}\}$$

Figure 18: Simplified query (not localized)

```
class DBData { public string Name; }

return ExecuteQuery<DBData>(
  @"SELECT Team.Name FROM Team
  INNER JOIN Task ON Team.Id = Task.TeamId
  INNER JOIN Client ON Task.CliId = Client.Id
  WHERE Task.Date = '8/5' GROUP BY Team.Name");
```

Figure 19: Code for Figure 11, using only top level data.

$$\tau = \langle\, name : \mathsf{String},\ details : \langle task : \langle title : \mathsf{String}\rangle,\ client : \langle name : \mathsf{String}\rangle,\ coords : \langle lat : \mathsf{Num}, lng : \mathsf{Num}\rangle\rangle^*\rangle^*$$

then we no longer can omit the call to the `GEO` service. Furthermore, the groupby operation needs to be performed in memory. The resulting code is shown in Figure 20.

Note that even though the client's address is not present in the output, it still needs to be fetched from the database because it is needed to invoke the `Coords` service. The algorithm accounts for data mentioned in arguments of data-sources, filters, join conditions and paths of in-place operations, and discards only the unnecessary parts.

8. Related Work

Unlike many DSLs for the development of complete applications [9, 11], we focus on the problem of typeful integration of data sources, as in [16], but dealing with the particular aspect of distributing and optimizing code given a usage.

Our proposal provides a flexible nesting base model (as [4]), that fits several variants of data repositories, from relational databases, to NoSQL document based repositories, to parameterizable web services. An approach like [5] may be used to complement our approach. Additionally, we naturally deal with raw nested data [8], by means of our in-place modification operation.

Our work is related to the composition of higher order queries, and higher order manipulation of XML data [1, 19]. We use the uniform and compositional mechanism of in-place modifications, that applies to all kinds of repositories, and is suitable to query simplification.

Capabilities of data repositories are captured using description logics, in systems that solve the problem of answering queries by combining existing repositories [22]. Our goal is different, as we limit the capabilities to the language operations, and do not use the semantics of the schema.

Related work includes systems that integrate, behind a single interface, several data based systems (e.g. [13]). We address a simpler, and yet relevant scenario, that is how to integrate data sources via a programming tool for applications, that typically are already capable of orchestrating several data sources. This approach lets the developer seamlessly access and combine data sources of different natures.

9. Final Remarks

We introduced a common data manipulation language for nested collections that allows the orchestration of several data sources remotely located. Our language abstracts the capabilities of each data repository, and the type based compilation and optimization algorithm we presented allows for the generation of specific code for each kind of database engine, eagerly aggregating the operations as close to the data sources as possible, and falling back to in-memory processing when needed.

Our model is the base for a new visual data manipulation language in the OutSystems platform, one that allows the gradual construction of queries with immediate feedback to developers. Future work includes the definition of the query rewriting mechanism that simplifies the deep data manipulation operations on nested data, and the corresponding integration with the localization algorithm.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric General-purpose Language. In *Proc. Int. Conference on Functional Programming*, 2003.

[2] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.-C. Wu. Distributed/heterogeneous query processing in microsoft sql server. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, 2005.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD RECORD*, 23, 1994.

[4] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.*, 149(1), Sept. 1995.

[5] J. Cheney, S. Lindley, and P. Wadler. Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets. In *Proc. of Int. Conference on Management of Data*, 2014.

[6] A. Chlipala. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, 2015.

[7] J. Clark and S. J. DeRose. XML Path Language (XPath) Version 1.0, 1999. URL `www.w3.org/TR/XPath`.

[8] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. In *Proc. of Int. Conference on Management of Data*, 1989.

```
class Client {
  public string Address;
  public string Name;
}
class Team {
  public string Name;
}
class Task {
  public string Title;
}
class DBData {
  public Team Team;
  public Task Task;
  public Client Client;
}
```

```
class Coordinate {
  public float Lat;
  public float Lng;
}

class Detail {
  public Client Client;
  public Task Task;
  public Coordinate Loc;
}

class MemData {
  public IEnumerable<Detail> Details;
  public string Name;
}
```

```
return ExecuteQuery<DBData>(
  @"SELECT Team.Name, Task.Title,
          Client.Address, Client.Name FROM Team
  INNER JOIN Task ON Team.Id = Task.TeamId
  INNER JOIN Client ON Task.CliId = Client.Id
  WHERE Task.Date = '8/5'")
.GroupBy(
    elem => new { Name = elem.Team.Name },
    elem => elem,
    (key, elems) => new MemData() {
      Name = key.Name,
      Details = elems.Select(a => new Detail() {
        Client = a.Client,
        Task = a.Task,
        Loc = GEO.Coords(a.Client.Address)
      })
   });
```

Figure 20: Generated code for query in Figure 11, using task's title, the client's name and the address' coordinates.

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *Proc. Int. Conference on Formal Methods for Components and Objects*, 2007.

[10] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 2001.

[11] Y. Fu, K. W. Ong, and Y. Papakonstantinou. Declarative Ajax Web Applications through SQL++ on a Unified Application State. In *Proceedings of Intern. Symposium on Database Programming Languages*, 2013.

[12] N. Grade, L. Ferrão, and J. C. Seco. Optimizing Data Queries Over Heterogeneous Sources. In *Proceedings of the 5th Simpósio de Informática*, Évora, Portugal, 2013.

[13] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: the teenage years. In *Proc. of int. conference on Very large data bases*, pages 9–16. VLDB Endowment, 2006.

[14] S. P. Jones and P. Wadler. Comprehensive Comprehensions. In *Proc. Haskell Workshop*, Haskell '07, 2007.

[15] João Costa Seco and Hugo Lourenço and Paulo Ferreira. A common data manipulation language. Technical report, Universidade Nova de Lisboa, 2015. URL `ctp.di.fct.unl.pt/~jcs/techreport-dbpl.pdf`.

[16] S. Lindley and J. Cheney. Row-based Effect Types for Database Integration. In *Proc. Workshop on Types in Language Design and Implementation*, pages 91–102, 2012.

[17] OutSystems. Using Aggregates - Fetching Data from the Database. Tech. Documentation, 2015. URL `www.outsystems.com`.

[18] Y. Papakonstantinou, A. Gupta, and L. M. Haas. Capabilities-based query rewriting in mediator systems. *Distributed and Parallel Databases*, 6(1), 1998.

[19] J. Robie et al. XQuery 3.0: An XML Query Language, 2014. URL `www.w3.org/TR/xquery-30/`.

[20] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Companion to the 21th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[21] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., 5 edition, 2006.

[22] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *The Journal of Logic Programming*, 43(1):75 – 122, 2000.