# Reorganizing Queries with Grouping

Rui Okura
University of Tsukuba
Tsukuba, Japan
rui@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

## Abstract

Language-integrated query has attracted much attention from researchers and engineers. It enables one to write a database query with high-level abstractions, which makes it possible to compose, iterate, and reuse queries. An important issue in language-integrated query is the N+1 query problem, and Cheney et al. proposed a program-transformation approach to solve it for a core language of Microsoft's LINQ. In our previous work, we extended their language to grouping (GROUP BY in SQL) and aggregate functions, and showed that any term can be transformed to a single SQL query. It still has a problem in that the resulting queries may be unnecessarily large and inefficient.

This paper solves the problem. Our key idea is re-organization of queries with nested control structures. While our previous work decomposes grouping into finer primitives before transformation, the new algorithm fuses nested control structures after transformation, while keeping the absence of nested data structures. Our algorithm also eliminates correlated subqueries as much as possible, to obtain better performance. We have conducted performance measurements, which shows that our new algorithm reduces the size of generated queries and improves the performance for several examples.

***CCS Concepts:*** • **Theory of computation → Database query processing and optimization (theory)**; • **Software and its engineering → Functional languages**.

***Keywords:*** database, language-integrated query, grouping, aggregation, normalization, type safety

**ACM Reference Format:**
Rui Okura and Yukiyoshi Kameyama. 2020. Reorganizing Queries with Grouping. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426960


*GPCE '20, November 16–17, 2020, Virtual, USA*

ACM ISBN 978-1-4503-8174-1/20/11…$15.00
https://doi.org/10.1145/3425898.3426960

## 1 Introduction

Language-integrated query provides high-level abstractions to the database queries, and has been attracting much attention from researchers and engineers. Studies based on typed functional languages provide type safety for language-integrated query, such Ohori et al's SML# [9], Grust et al's Ferry [5], and Cooper et al's Links [4]. Among all, one of the most popular system is Microsoft's LINQ[1], which integrates SQL with a high-level programming language F# [8].

Research on language-integrated query has been carried out by a number of studies. Earlier approaches were conservative in the sense that they put syntactic restrictions on its query sublanguage so that only SQL-equivalent queries can be written in the language, which leads to poor composability and reusability [6]. More ambitious approaches including Cooper [3] and Cheney et al. [1] allow arbitrary composition of program fragments, thus allowing component-based construction of large complicated queries. By composing queries, we naturally get a query with nested control structures (such as SELECT in SQL), and nested data structures (such as record of records, record of list of records), which are not directly implementable by most dialects of SQL. Cheney et al. introduced a quotation mechanism for queries, and program transformation over quoted terms to eliminate nested data/control structures. They succeeded in transforming any query into a single SQL query, but their target SQL does not contain grouping (GROUP BY) and aggregate functions (such as MAX and SUM), which are particularly important in realistic applications.

In our previous work [10] we proposed an algorithm to cover grouping and aggregate functions under the assumption that the target SQL allows subqueries (nested queries). [2]

$$
\begin{aligned}
& \overbrace{g_2}\ \circ\ f_0\ \circ\ \overbrace{g_1} \\
\rightsquigarrow\ & \overbrace{f_4 \circ \mathbf{G}_2 \circ f_3}\ \circ\ f_0\ \circ\ \overbrace{f_2 \circ \mathbf{G}_1 \circ f_1} && \text{(decomposition)} \\
\rightsquigarrow\ & f_4 \circ \mathbf{G}_2\ \circ\ f_5\ \circ\ \mathbf{G}_1 \circ f_1 && \text{(normalization)}
\end{aligned}
$$

**Figure 1.** Overview of the Previous Work

[1] https://docs.microsoft.com/en-us/dotnet/csharp/linq/
[2] Subqueries are those queries whose input, output, or guard conditions contain another query. PostgreSQL and several dialects of SQL allows subqueries.

Fig. 1 illustrates how the algorithm works. The goal of this algorithm is to eliminate nested data structures from a query with grouping and aggregate functions.

Terms such as $f_i$ and $g_i$ denote lambda terms $\lambda x.Q$ (a query which takes an input table $x$ and computes the value of $Q$), and $\circ$ denotes function composition. Suppose we have a query $g_2 \circ f_0 \circ g_1$ where $g_i$ are queries with grouping and $f_0$ is a query without grouping. Functions in our language may pass *nested data structures* such as a multi-set (bag) of multi-sets of records[3] as intermediate data, hence the output of $g_1$ (the input of $f_0$) may be nested data, and similarly for the output of $f_0$.

In our previous work, we found that, by decomposing a query with grouping ($g_1$ and $g_2$) into composition of smaller operations such as a new grouping primitive **G** and existing primitives in Cheney et al.'s language, we can apply Cheney et al.'s normalization rules to eliminated nested data. The primitive **G** is the 'minimum' grouping operator, namely, it cannot destruct its input, nor compose output, and only does grouping by specified keys and applying aggregate functions to several fields of data. In the above figure, $g_1$ is decomposed to $f_2 \circ \mathbf{G}_1 \circ f_1$ where $\mathbf{G}_1$ is an instance of **G** and $f_i$ are instances of existing primitives. Then, we can apply normalization rules by Cheney et al. to $f_i \circ f_j$. We proved that, even with the **G** operator, we can eliminate all nested data structures, and the resulting term can always be transformed to a single query in SQL which allows subqueries.

Unfortunately, there is a problem in our previous work, which we address in this paper. The final term $f_4 \circ \mathbf{G}_2 \circ f_5 \circ \mathbf{G}_1 \circ f_1$ is converted to a query which has nested subqueries of depth five (five times nesting of SELECT statements), each of which corresponds to one of $f_i$ and $\mathbf{G}_i$. This is unnecessarily redundant compared with the term before the decomposition, whose depth is only three.

$$
\begin{array}{lcccccc}
 & g_2 & \circ & f_0 & \circ & g_1 \\
\rightsquigarrow & f_4 \circ \mathbf{G}_2 \circ f_3 & \circ & f_0 & \circ & f_2 \circ \mathbf{G}_1 \circ f_1 \\
\rightsquigarrow & \underbrace{f_4 \circ \mathbf{G}_2 \quad \circ \quad f_5}_{\textbf{select\_g}} & \circ & & & \underbrace{\mathbf{G}_1 \circ f_1}_{\textbf{select\_g}} \\
\rightsquigarrow & \textbf{select\_g} & \circ & & & \textbf{select\_g}
\end{array}
$$

**Figure 2.** Overview of This Work

Fig. 2 gives our solution for the above problem. The idea is simple; while the decomposed form is needed to apply normalization, it is not needed after all nested data structures are eliminated. Hence, we *re-organize* the resulting query by fusing function composition as much as possible, reducing the size of the query. For this purpose, we present several fusion rules, which introduce no new nested data. For instance, $\mathbf{G} \circ f_i$ is fused to a single function and converted to a SELECT clause with grouping (written **select_g** in the figure). We also found a rule for eliminating correlated subqueries, which, if applicable, drastically improves the performance of generated queries.

We have conducted performance measurements for various queries in language-integrated query with nested data structures and nested control structures. Combined with the algorithm in our previous work, the new algorithm works surprisingly well; the performance of SQL queries generated by our algorithm is the same or better than the performance of those in our previous work for all examples, and it is always better than those by Microsoft's LINQ.

The contribution of this paper is summarized as follows:

- We show that the algorithm in our previous work sometimes generates large queries for some cases, and their performance is rather poor if they contain correlated subqueries.
- We introduce new transformation rules which fuse nested control structures in the generated queries.
- We conduct performance measurements on various examples which use grouping and aggregation. The results are in favor of our research compared with Microsoft's LINQ and our previous work.

The rest of this paper is organized as follows. Section 2 informally explains the problem in our previous work and the overview of the results in this paper using examples. Section 3 gives the source language, its type system, and program transformations in the previous work. Section 4 presents our re-organization algorithm for queries with grouping and aggregate functions, and Section 5 gives the final transformation to SQL. Section 6 explains several involved examples. The performance measurements for our language is explained in Section 7. Section 8 gives conclusion.

## 2 Examples

This section informally explains the outline of our work using several examples. We use the database consisting of two tables in Fig. 3. The products table has the fields[4] for product

products

| pid | name | cat | price |
|---|---|---|---|
| 1 | shirt | 110 | 100 |
| 2 | T-shirt | 110 | 200 |
| 3 | pants | 111 | 500 |
| 4 | suit | 210 | 1000 |

orders

| date | pid | qty |
|---|---|---|
| 2020-01-01 | 1 | 3 |
| 2020-01-01 | 1 | 10 |
| 2020-01-01 | 2 | 2 |
| 2020-01-02 | 1 | 5 |
| 2020-01-02 | 4 | 15 |
| 2020-01-02 | 4 | 20 |

**Figure 3.** Sample database tables

[3] A simple (non-nested) data of table type is a multi-set of records which consist of basic data such a strings and numbers. Most SQL dialects can manipulate such data only.

[4] We call each element of a record a field, which means a column for a database table.

ID (pid), name (name), category (cat), and price (price), and the orders table has the fields for the date of order (date), product ID (pid), and quantity (qty).

### 2.1 First Example

The following SQL query computes, for each order ID, the sum of quantities ordered on 2020-01-01.

```
SELECT o.pid AS pid,
       SUM(o.qty) AS qty_sum
FROM orders AS o
WHERE o.date = "2020-01-01"
GROUP BY o.pid
```

Here, the SELECT statement is used with GROUP BY, and classifies the given data by the value of the key o.pid. SUM is an aggregate function which computes the sum of all data in each group. A query with grouping and aggregate functions is frequently used in practical applications of database, yet, it has been an open problem [2] to cover GROUP BY in language-integrated query before our previous study.

In our previous work, we proposed a solution for this problem when the target SQL allows subqueries. Our idea is *decomposition* of queries, and the above query can be expressed in the following decomposed form:

$$\begin{aligned} Q(\textbf{simple}) = \mathcal{G}_{(\text{pid},\alpha)}(&\textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\ &\textbf{where } (o.\text{date} = \text{“2020-01-01”}) \\ &\textbf{yield } o) \\ \text{where } \alpha &= \{(\text{qty}, \text{SUM}, \text{qty\_sum})\} \end{aligned}$$

Here, $\textbf{for}(x \leftarrow T)$ **where** $B$ **yield** $M$ corresponds to `SELECT M FROM T AS x WHERE B` in SQL. The $\mathcal{G}$-operator is our new operator for grouping and aggregate functions. It takes two parameters (pid and $\alpha$ in the above example) and an argument (the term starting from **for**). Its first parameter is the field(s) to be used as grouping key(s), and the second one specifies (1) the field to which an aggregate function is applied, (2) an aggregate function, and (3) the field name which stores the result of aggregation. The term $\mathcal{G}_{(\text{pid},\alpha)}(N)$ classifies the input table $N$ by the value of pid, computes the sum of the field qty in each group, and returns a bag (multi-set) of the records consisting of the value of pid and the sum (in the field qty_sum), hence, $Q(\textbf{simple})$ does the same computation as the above SQL query.

The $\mathcal{G}$-operator is a rather simplified operator than the grouping feature in SQL. For instance, we cannot compute SUM(qty * qty), nor SUM(qty)/2 by $\mathcal{G}$ alone. However, we can express any SELECT statement with GROUP BY using our $\mathcal{G}$-operator and existing primitives; see Section 3.

Since the query $Q(\textbf{simple})$ uses no nested data as intermediate data, we can immediately convert it a SQL query as follows.

```
SELECT x.pid AS pid,
       SUM(x.qty) AS qty_sum
FROM (SELECT o.*
      FROM orders AS o
      WHERE o.date = "2020-01-01") AS x
GROUP BY x.pid
```

We see a problem in our previous work here. Initially we had a simple query without nesting, but after transformation we got a larger SQL query with nested SELECT statements. In general, the query generated by our previous transformation can be up to three times as large as the query before transformation, which is unacceptable. Even worse, the large size of queries sometimes causes a serious performance problem of generated queries. This paper fixes these problems.

### 2.2 Second Example

The second example is an instance of a nested query $g_2 \circ f_1 \circ g_1$ illustrated in the previous section.

Let $g_1$ be the following function.

$$\begin{aligned} g_1(t_1, t_2) = \textbf{for}(&x \leftarrow \mathcal{G}_{(\{\text{date},\text{pid}\},\alpha_1)}(t_1)) \\ &\textbf{yield } \{\text{pid} = x.\text{pid}, \\ &\quad \text{sales} = \textbf{for}(p \leftarrow t_2) \\ &\qquad \textbf{where } (x.\text{pid} = p.\text{pid}) \\ &\qquad \textbf{yield } \{\text{category} = p.\text{cat}, \\ &\qquad\quad \text{sale} = p.\text{price} * x.\text{qty\_sum}\}\} \\ &\text{where } \alpha_1 = \{(\text{qty}, \text{SUM}, \text{qty\_sum})\} \end{aligned}$$

We can obtain a concrete query by applying $g_1$ to two tables, for instance, $g_1(\textbf{table}(\text{”orders”}), \textbf{table}(\text{”products”}))$. $g_1$ first executes $\mathcal{G}_{(\{\text{date},\text{pid}\},\alpha_1)}(t_1)$ which computes a bag of the sum of quantity in each group classified by the value of the *date* field and the *pid* field in the table $t_1$. Then it computes a nested data consisting of a bag of records with the fields pid and sales, and the sales field is a bag of records.

The next function $f$ extracts the sales field of an input $t$, which is assumed to be a bag of records, and does some computation for each record.

$$\begin{aligned} f(t) = \textbf{for}(&y \leftarrow t) \\ &\textbf{for}(z \leftarrow y.\text{sales}) \\ &\textbf{yield } \{\text{category} = z.\text{category}, \\ &\quad \text{sale} = z.\text{sale} * 0.8\} \end{aligned}$$

The third function in this series is $g_2$, which computes the sum of sales grouped by category, and extracts summation of sales (sales_sum) and multiplies it by 100.

$$\begin{aligned} g_2(t) = \textbf{for}(&v \leftarrow \mathcal{G}_{(\text{category},\alpha_2)}(t)) \\ &\textbf{yield } \{\text{result} = v.\text{sale\_sum} * 100\} \\ &\text{where } \alpha_2 = \{(\text{sale}, \text{SUM}, \text{sale\_sum})\} \end{aligned}$$

We can compose the three functions and apply the result to concrete tables to obtain a concrete query. By executing $(g_2 \circ f \circ g_1)(\textbf{table}(\text{”orders”}), \textbf{table}(\text{”products”}))$, we get:

$$[\{\text{result} = 176000\}; \{\text{result} = 2800000\}].$$

Although the input and the output of this query are non-nested data (a bag of records which consists of fields with basic values), nested data structures (a bag of records of bags) are passed around between adjacent functions, hence the above query is not directly expressible in SQL.

The transformation in our previous work can eliminate the nested data structures in this query, and generate the following SQL query (which we call $Q_0$).

```
SELECT v.sale_sum * 100 AS result
FROM (SELECT z.category AS category,
             SUM(z.sale) AS sale_sum
      FROM (SELECT p.cat AS category,
                   p.price * x.qty_sum *
                       0.8 AS sale
            FROM (SELECT y.date AS date,
                         y.pid AS pid,
                      SUM(y.qty) AS qty_sum
                  FROM (SELECT o.*
                        FROM orders AS o)
                           AS y
                  GROUP BY y.date,
                           y.pid) AS x,
                  products AS p
            WHERE x.pid = p.pid) AS z
      GROUP BY z.category) AS v
```

The result is a single SQL, and can be executed in SQL processors which allow subqueries. However, it contains four-times nested subqueries, which is unnecessarily large. In general, the decomposition may triple the depth of subqueries of queries, which is not acceptable. It also affects the performance of generated queries. It is true that a SQL optimizer sometimes improves the performance of nested queries, but it is not always the case. We found several examples whose performance is rather poor compared with a hand-written, equivalent SQL query.

The present paper solves this problem by transforming the generated queries further. The above example is transformed by our new transformation to the following one.

```
SELECT SUM(z.sale) * 100 AS result
FROM (SELECT p.cat AS category,
             p.price * SUM(o.qty) * 0.8
                                 AS sale
      FROM orders AS o, products AS p
      WHERE o.pid = p.pid
      GROUP BY o.date, o.pid, p.pid,
               p.name, p.cat,
               p.price) AS z
GROUP BY z.category
```

The depth of subqueries in this query is smaller than that of the query before transformation. By executing it, we get the same result as the latter.

In the next section we formally introduce the language and our new transformation rules.

## 3 The Language with Grouping

This section presents the language Quelg, its type system, and program transformation rules in our previous work [10]. The new transformation in this paper will be given in the next section.

### 3.1 Language

We define the language Quelg for language-integrated query. It is based on Cooper's source language [3] without effects (which is nearly the same as Nested Relational Calculus [13]), and Cheney et al.'s T-LINQ [1] restricted to the code-level terms. We added the grouping operator and aggregate functions to their language.

Fig. 4 defines the syntax of terms in Quelg.

$$
\begin{array}{llll}
\text{Terms} & M, N & ::= & x \mid c \mid \lambda x.\, M \mid M\ N \mid \oplus(\overline{M}) \\
 & & & \mid M \uplus N \mid \mathbf{for}(x \leftarrow M)\ N \\
 & & & \mid \mathbf{where}\ M\ N \mid \mathbf{yield}\ M \\
 & & & \mid [\,] \mid \mathbf{table}(t) \\
 & & & \mid \{\overline{l = M}\} \mid M.l \mid \mathcal{G}_{(\kappa,\alpha)}(M) \\
\text{Spec} & \alpha & ::= & \{\overline{(l, \circledcirc, l')}\}
\end{array}
$$

**Figure 4.** Syntax of Terms in Quelg

Terms are either a variable ($x$), constant ($c$), $\lambda$-abstraction, function application, primitive function call (where $\oplus$ denotes a primitive function), multiset union ($M \uplus N$), comprehension ($\mathbf{for}(x \leftarrow M)\ N$), conditional ($\mathbf{where}\ M\ N$), singleton ($\mathbf{yield}\ M$), empty bag ($[\,]$), table expression ($\mathbf{table}(t)$), record construction ($\{\overline{l = M}\}$), selection ($M.l$), or grouping ($\mathcal{G}_{(\kappa,\alpha)}(L)$).

An overlined expression denotes a sequence, for instance, $\overline{l = M}$ is an abbreviation for $l_1 = M_1, \cdots, l_n = M_n$ where the length $n$ is implicit. The metavariable $t$ denotes a table name (given by a string), $l$ is a field name in the record, and $\kappa$ is a finite set of field names used as grouping keys. The specification $\alpha$ is a finite set of triples consisting of a field name ($l$) for the target of aggregation, an aggregate function ($\circledcirc$), and a field name ($l'$) for the result of aggregation.

Let us briefly explain the semantics of important constructs. First, most expressions compute a bag as its value, where a bag is a multiset (an unordered list), and we write it using list notation such as $[a, a, b, c]$. A table is a bag of records such as $\{age = 25, name = \text{"}Tom\text{"}\}$. Whereas in the standard database, each field of a record in a table should hold a basic value such as integers or strings, Quelg allows an arbitrarily data as the value of a field, hence nested data structures such as a bag of records of records are legitimate (typable) in Quelg.

The term $\mathbf{for}(x \leftarrow M)\ N$ is bag-comprehension; for each element of the value of $M$ (which should be a bag), it computes the term $N$ which should return a bag, and returns the multiset-union of the bags. The term $\mathbf{where}\ L\ M$ returns the value of $M$ if $L$ returns true, and returns the empty bag [ ] otherwise. The term $\mathbf{yield}\ M$ returns a singleton multiset consisting of the value of $M$.

The term $\mathcal{G}_{(\kappa,\alpha)}(M)$ expresses the minimum functionality of grouping and aggregation. It gets as input the value of $M$ (which should be a bag), and classifies the records based on the values of grouping keys specified by $\kappa$, aggregates the values in the same group, and finally returns a bag of records consisting of grouping keys and aggregated values.[5] For instance, by computing the query $\mathcal{G}_{(\text{cat},\alpha)}(\mathbf{table}(\text{"products"}))$ where $\alpha = \{(\text{price}, \text{MAX}, \text{max\_price})\}$ for the products table in Section 2, we get $[\{\text{cat} = 210, \text{max\_price} = 1000\}]$ as its result. We use typical aggregate functions such as SUM, AVG, MAX, MIN and COUNT in the examples, but as long as our transformation is concerned, any aggregate functions are allowed if its input and output are basic types. See the typing rule for grouping.

We have the standard notion of variable binding; the variable $x$ is bound in the subterm $M$ of $\lambda x.\ M$ and $\mathbf{for}(x \leftarrow L)\ M$. We identify $\alpha$-equivalent terms, and substitution used in program transformation should be the capture-avoiding one.

Although our grouping operator has limited functionality, we can express the SELECT statement with the GROUP BY clause in SQL by combining our primitives with others. For instance, the following SQL query

```
SELECT p.cat as cat,
       SUM(p.price * o.qty) as sales
FROM   products AS p, orders AS o
WHERE  p.pid = o.pid
GROUP BY p.cat
HAVING SUM(p.price * o.qty) > 2000
```

can be expressed as a term in Quelg as follows:

$$\begin{aligned}
&\mathbf{for}(r \leftarrow \mathcal{G}_{(\text{cat},\alpha)}(\mathbf{for}(p \leftarrow \mathbf{table}(\text{"products"}))\\
&\qquad\qquad \mathbf{for}(o \leftarrow \mathbf{table}(\text{"orders"}))\\
&\qquad\qquad \mathbf{where}\ (p.\text{pid} = o.\text{pid})\\
&\qquad\qquad \mathbf{yield}\ \{\text{cat} = p.\text{cat}, \text{sales} = p.\text{price} * o.\text{qty}\}))\\
&\mathbf{where}\ (r.\text{s} > 2000)\\
&\mathbf{yield}\ \{\text{cat} = r.\text{cat}, \text{sales} = r.\text{s}\}\\
&\text{where}\ \alpha = \{(\text{sales}, \text{SUM}, \text{s})\}
\end{aligned}$$

Note that our operator $\mathcal{G}$ does grouping and aggregation only, and all other works are done by existing primitives. We can thus *decompose* a primitive for grouping into the combination of $\mathcal{G}$ and existing primitives. By this decomposition, the term will have more opportunity to be transformed, which is the key to our previous work.

[5] For simplicity, we assume that all grouping keys are contained in the results. See the typing rule.

### 3.2 Type System

The language Quelg is a statically typed language, and all transformation rules must preserve typing of terms. Since it has been argued in existing works in depth, we give a brief overview of the type system.

Fig. 5 defines types and typing environments.

| | | | |
|---|---|---|---|
| Base types | $O$ | ::= | $\text{Int} \mid \text{Bool} \mid \text{String}$ |
| Types | $A, B$ | ::= | $O \mid A \to B \mid \text{Bag}\ A \mid \{\overline{l : A}\}$ |
| Env | $\Gamma$ | ::= | $\cdot \mid \Gamma, x : A$ |

**Figure 5.** Types of Quelg

A type is a basic type, a function type, a bag type Bag $A$ for multisets, or a record type $\{\overline{l : A}\}$, which is abbreviation of $\{l_1 : A_1, \cdots, l_k : A_k\}$. Throughout this paper, we assume that all field names $l_1, \cdots, l_k$ are mutually distinct.

For basic types $O_i$, the type Bag $\{\overline{l : O}\}$ is called a table type. Standard SQL can manipulate the values of table type only, and other nested data such as a bag of bags of records must be translated away before generating an executable SQL. A typing context $\Gamma$ is standard where $\cdot$ denotes the empty environment.

Fig. 6 shows typing rules for important constructs.

$$\text{FOR}\ \frac{\Gamma \vdash M : \text{Bag}\ A \quad \Gamma, x : A \vdash N : \text{Bag}\ B}{\Gamma \vdash \mathbf{for}(x \leftarrow M)\ N : \text{Bag}\ B}$$

$$\text{WHERE}\ \frac{\Gamma \vdash L : \text{Bool} \quad \Gamma \vdash M : \text{Bag}\ A}{\Gamma \vdash \mathbf{where}\ L\ M : \text{Bag}\ A} \qquad \text{SINGLETON}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{yield}\ M : \text{Bag}\ A}$$

$$\text{GROUPING}\ \frac{\begin{array}{c}\Gamma \vdash L : \text{Bag}\ \{\overline{\kappa : O''}, \overline{m : O}\} \quad \overline{l} \subseteq \overline{m} \\ \alpha = \{(\overline{l}, \circledcirc, \overline{l'})\} \quad \circledcirc_i : O_j \to O'_j\ (\text{for all } l_i \text{ and } m_j \text{ s.t. } l_i = m_j)\end{array}}{\Gamma \vdash \mathcal{G}_{(\kappa,\alpha)}(L) : \text{Bag}\ \{\overline{\kappa : O''}, \overline{l' : O'}\}}$$

**Figure 6.** Type System of Quelg

The last typing rule is the one for the grouping term $\mathcal{G}_{(\kappa,\alpha)}(L)$. The first condition in its hypothesis says $L$ must have a table type. $\overline{l} \subseteq \overline{m}$ means that the former set is a subset of the latter, in other words, for any $l_i$, there exists an $m_j$ such that $l_i = m_j$. The last condition in the hypothesis says that the $i$-th aggregate function $\circledcirc_i$ must have the type $O_j \to O'_j$. Then $\mathcal{G}_{(\kappa,\alpha)}(L)$ returns a bag of records which has

the values of all the grouping keys, and the results of applying aggregate functions to the specified field in $\alpha$ with the new field name $l'$.

Note that our operator $\mathcal{G}$ does grouping and aggregation only, and all other works are done by existing primitives in Cheney et al.'s language. We can thus *decompose* SQL's big primitive for grouping into the combination of $\mathcal{G}$ and existing primitives. Since **for** and other existing primitives can be transformed by Cheney et al.'s transformation rules, the term in the above form has more opportunity to be transformed, which is the key to our previous work.

### 3.3 Normalization

The query avalanche problem, or the $N+1$-query problem in language-integrated query means that, to execute a nested **for**-term using the SQL processor which does not allow subqueries, one would need to generate many SQL queries and interact with SQL processor many times, which degrades the performance. Cooper solved the problem by program transformation [3], and Cheney et al. formalized his idea in a program-generation framework; they introduced a typed two-level language, and proved that any closed terms of table types can be transformed to normal form which is directly translated to SQL. Our previous work extended it to a language with grouping and aggregate functions. Surprisingly, no new transformation rules are needed other than decomposition of a big grouping primitive. By applying their transformation to our language, we have solved the $N+1$-query problem for our language.

Fig. 7 shows the normalization rules by Cheney et al.

The rules are classified into two stages, where Stage 1 rules should be applied before Stage 2 rules. [6] The former consists of structural reductions for various types, while the latter consists of ad hoc simplification rules. More detailed explanation can be found in the literature [1].

It is proved that these rules are normalizing [10], and the syntax of normal form is given in Fig. 8.

$U$ is the top-level category, namely, a closed term of table type in Quelg is normalized to $U$. Compared with the normal form in Cheney et al.'s work, our normal form allows $\mathcal{G}_{(\kappa,\alpha)}(U)$ to appear in $H$, hence a term with nested control structures such as $\mathbf{for}(x \leftarrow \mathcal{G}_{(\kappa,\alpha)}(U))\ F$ is in normal form. Nevertheless, we can prove that, no nested data types (other than the table types) appear in the types of any subterms of normal form, hence we can convert normal form in Fig. 8 to a SQL query if subqueries are allowed. See Okura et al. [10] for details. The query $Q_0$ in Section 2 has been obtained in this way.

[6] The order of applying rules in each stage does not matter, as the rules are (semantically) confluent and strongly normalizing.

**(Stage 1)**

$$
\begin{array}{rcl}
(\lambda x.N)\ M & \rightsquigarrow & N[x := M] \\
\{\overline{l = M}\}.l_i & \rightsquigarrow & M_i \\
\mathbf{for}(x \leftarrow \mathbf{yield}\ M)\ N & \rightsquigarrow & N[x := M] \\
\mathbf{for}(x \leftarrow \mathbf{for}(y \leftarrow L)\ M)\ N & \rightsquigarrow & \\
\quad \mathbf{for}(y \leftarrow L)\ (\mathbf{for}(x \leftarrow M)\ N)\ (\text{if } y \notin FV(N)) & & \\
\mathbf{for}(x \leftarrow \mathbf{where}\ L\ M)\ N & \rightsquigarrow & \\
\quad \mathbf{where}\ L\ (\mathbf{for}(x \leftarrow M)\ N) & & \\
\mathbf{for}(x \leftarrow [])\ N & \rightsquigarrow & [] \\
\mathbf{for}(x \leftarrow M_1 \uplus M_2)\ N & \rightsquigarrow & \\
\quad (\mathbf{for}(x \leftarrow M_1)\ N) \uplus (\mathbf{for}(x \leftarrow M_2)\ N) & & \\
\mathbf{where\ true}\ M & \rightsquigarrow & M \\
\mathbf{where\ false}\ M & \rightsquigarrow & []
\end{array}
$$

**(Stage 2)**

$$
\begin{array}{rcl}
\mathbf{for}(x \leftarrow M)\ (N_1 \uplus N_2) & \hookrightarrow & \\
\quad (\mathbf{for}(x \leftarrow M)\ N_1) \uplus (\mathbf{for}(x \leftarrow M)\ N_2) & & \\
\mathbf{for}(x \leftarrow M)\ [] & \hookrightarrow & [] \\
\mathbf{where}\ L\ (M \uplus N) & \hookrightarrow & \\
\quad (\mathbf{where}\ L\ M) \uplus (\mathbf{where}\ L\ N) & & \\
\mathbf{where}\ L\ (\mathbf{where}\ M\ N) & \hookrightarrow & \\
\quad \mathbf{where}\ (L \wedge M)\ N & & \\
\mathbf{where}\ L\ (\mathbf{for}(x \leftarrow M)\ N) & \hookrightarrow & \\
\quad \mathbf{for}(x \leftarrow M)\ \mathbf{where}\ L\ N & &
\end{array}
$$

**Figure 7.** Normalization rules

$$
\begin{array}{llcl}
\text{Query} & U & ::= & U_1 \uplus U_2 \mid [\,] \mid F \\
\text{Comprehension} & F & ::= & \mathbf{for}(x \leftarrow H)\ F \mid Z \\
 & H & ::= & \mathbf{table}(t) \mid \mathcal{G}_{(\kappa,\alpha)}(U) \\
\text{Body} & Z & ::= & \mathbf{where}\ B\ Z \mid \mathbf{yield}\ R \mid H \\
\text{Record} & R & ::= & \{\overline{l = B}\} \mid x \\
\text{PrimitiveOp.} & B & ::= & \oplus(\overline{B}) \mid x.l \mid c \\
 & \alpha & ::= & \{\overline{(l, \circledcirc, l')}\} \\
\text{Key} & \kappa & ::= & \overline{l}
\end{array}
$$

**Figure 8.** Normal form of Quelg

## 4 Reorganizing Queries

This section introduces novel transformations to optimize normal form in the previous section, to reduce the size of queries and get better performance.

As we explained in Section 2, SQL queries obtained by our previous work is often unnecessarily large, and sometimes very inefficient. This is contrasting with the normal form by Cheney et al.'s work which always generates compact and efficient SQL queries. The difficulty in our language lies in the fact that the grouping operator itself allows no transformations at all, therefore our normal form ought to contain arbitrarily many nested control structures. Even

worse, no further transformation seems to be applicable to normal form within Quelg.

However, the normal forms are not arbitrarily complicated. We only have to consider the combination of $g \circ f$ or $f \circ g$, where $f$ is a **for**-term and $g$ is an instance of the grouping operator. Although fusing them within Quelg is not possible, fusing them in SQL is possible, as SQL's SELECT statements with GROUP BY is very expressive. Put differently, in our previous work, we decomposed SQL's grouping operation before program transformation, and in this work, we do its inverse; we *reorganize* (re-compose) the function composition such as $g \circ f$ into a single control structure, to obtain smaller queries in SQL. We also consider a fusion transformation for other patterns to further optimize queries.

### 4.1 Intermediate Representation

We introduce a language for intermediate representation (IR) which is useful to state our optimization for reorganization.

$$
\begin{array}{lrcl}
\text{Query} & Q & ::= & \textbf{\textit{select}}(I,\ O,\ C) \\
 & & \mid & \textbf{\textit{gselect}}(I,\ O,\ C,\ K,\ C) \\
 & & \mid & \textbf{\textit{unionall}}(Q,\ Q) \\
\text{Table} & T & ::= & \textbf{\textit{table}}(t) \mid \textbf{\textit{sub}}(\mathrm{Q}) \\
\text{Primitive} & B & ::= & c \mid x.l \mid \oplus(B) \\
\text{Input} & I & ::= & [\overline{(x \leftarrow T)}] \\
\text{Output} & O & ::= & \{\overline{l = B}\} \\
\text{Condition} & C & ::= & B \mid C \wedge C \\
\text{Key} & K & ::= & [\overline{B}]
\end{array}
$$

**Figure 9.** Syntax of Intermediate Representation

Fig. 9 gives the syntax of IR, which is a compact representation of SQL with small difference. The term ***select***$(I,\ O,\ C)$ corresponds to a SELECT statement without grouping:

**SELECT** $O$ **FROM** $I$ **WHERE** $C$

Similarly, the term ***gselect***$(I,\ O,\ C_1,\ K,\ C_2)$ corresponds to one with grouping as follows:

**SELECT** $O$ **FROM** $I$ **WHERE** $C_1$
**GROUP BY** $K$ **HAVING** $C_2$

$I$ and $O$ of these terms are called its input and output, respectively. The term ***unionall*** corresponds to UNION ALL in SQL. ***table***$(t)$ represents a table, and ***sub***$(Q)$ is a subquery.

The input $I$ of a query is an ordered list of variable-binding in the form $(x_i \leftarrow T_i)$ which will be translated to $T_i$ **AS** $x_i$ in SQL. For brevity, we use the bag-notation $[e_1; \cdots, e_n]$ for a list if it is used as an input term in IR. Variables in $I$ are bound by the ***select*** and ***gselect*** terms, but some care is needed to understand their scopes since the order of the elements in $I$ matters. To explain it using the term ***select***$([(x_1 \leftarrow T_1);\ (x_2 \leftarrow T_2)],\ O,\ C)$, the variable $x_1$ is bound in $T_2$, $O$ and $C$, while $x_2$ is bound in $O$ and $C$. Similarly to Quelg, we identify

$$
\begin{array}{rcl}
\mathcal{M}[\![U_1 \uplus U_2]\!] & = & \textbf{\textit{unionall}}(\mathcal{M}[\![U_1]\!],\ \mathcal{M}[\![U_2]\!]) \\
\mathcal{M}[\![[\,]]\!] & = & \textbf{\textit{select}}([\,],\ [\,],\ \textbf{false}) \\
\mathcal{M}[\![\mathcal{G}_{(\overline{\kappa},\alpha)}(U)]\!] & = & \\
\multicolumn{3}{l}{\quad \begin{cases}
\textbf{\textit{gselect}}([(y \leftarrow \textbf{\textit{table}}(t))],\ \{\overline{l' = \circledcirc(y.l)}\}, & \\
\qquad \textbf{true},\ [\overline{y.\kappa}],\ \textbf{true}) & \textit{if } U = \textbf{table}(t) \\
\textbf{\textit{gselect}}([(y \leftarrow \textbf{\textit{sub}}(\mathcal{M}[\![U]\!]))],\ \{\overline{l' = \circledcirc(y.l)}\}, & \\
\qquad \textbf{true},\ [\overline{y.\kappa}],\ \textbf{true}) & \textit{otherwise} \\
(\textit{where } \alpha = \{(\overline{l, \circledcirc, l')}\}) &
\end{cases}} \\
\mathcal{M}[\![\textbf{for}(x \leftarrow H)\ F]\!] & = & \\
\multicolumn{3}{l}{\quad \begin{cases}
\textbf{\textit{select}}((x \leftarrow \mathcal{M}^{aux}[\![H]\!]) :: I,\ O,\ C) & \\
\qquad \textit{if } \mathcal{M}[\![F]\!] = \textbf{\textit{select}}(I,\ O,\ C) & \\
\textbf{\textit{gselect}}((x \leftarrow \mathcal{M}^{aux}[\![H]\!]) :: I,\ O,\ C_1,\ [\overline{x.l}] \mathbin{@} K,\ C_2) & \\
\qquad \textit{if } \mathcal{M}[\![F]\!] = \textbf{\textit{gselect}}(I,\ O,\ C_1,\ K,\ C_2) &
\end{cases}} \\
\mathcal{M}[\![\textbf{where } B\ Z]\!] & = & \\
\multicolumn{3}{l}{\quad \begin{cases}
\textbf{\textit{select}}(I,\ O,\ C \wedge \mathcal{M}[\![B]\!]) & \\
\qquad \textit{if } \mathcal{M}[\![Z]\!] = \textbf{\textit{select}}(I,\ O,\ C) & \\
\textbf{\textit{gselect}}(I,\ O,\ C_1 \wedge \mathcal{M}[\![B]\!],\ K,\ C_2) & \\
\qquad \textit{if } \mathcal{M}[\![Z]\!] = \textbf{\textit{gselect}}(I,\ O,\ C_1,\ K,\ C_2) &
\end{cases}} \\
\mathcal{M}[\![\textbf{table}(t)]\!] & = & \\
\multicolumn{3}{l}{\quad \textbf{\textit{select}}([(y \leftarrow \textbf{\textit{table}}(t))],\ \{\overline{l = y.l}\},\ \textbf{true})} \\
\mathcal{M}[\![\textbf{yield } R]\!] & = & \textbf{\textit{select}}([\,],\ \mathcal{M}[\![R]\!],\ \textbf{true}) \\
\mathcal{M}[\![\{\overline{l = B}\}]\!] & = & \{\overline{l = \mathcal{M}[\![B]\!]}\} \\
\mathcal{M}[\![\oplus(\overline{B})]\!] & = & \oplus(\overline{\mathcal{M}[\![B]\!]}) \\
\mathcal{M}[\![e]\!] & = & e \textit{ for } e = x,\ c,\ x.l \\
\mathcal{M}^{aux}[\![H]\!] & = & \\
\multicolumn{3}{l}{\quad \begin{cases}
\textbf{\textit{table}}(t) & \textit{if } H = \textbf{table}(t) \\
\textbf{\textit{sub}}(\mathcal{M}[\![\mathcal{G}_{(\overline{\kappa},\alpha)}(U)]\!]) & \textit{if } H = \mathcal{G}_{(\overline{\kappa},\alpha)}(U)
\end{cases}}
\end{array}
$$

**Figure 10.** Translation from Quelg to IR

two $\alpha$-equivalent terms, and bound variables are renamed if name clash occurs during program transformation.

Precise correspondence between IR and SQL will be given later as a translation from the former to the latter.

Fig. 10 defines the translation from normal form in Fig. 8 to IR. Here $t_1 :: t_2$ is the cons operation for lists, and $s_1 \mathbin{@} s_2$ is concatenation for sequences. The variable $y$ used in the results of the translation should be a fresh variable.

Let us explain the translation, which converts any normal form of bag type into a ***select***-term or a ***gselect***-term. This allows us to treat IR terms uniformly.

Our intention of the translation is the 'output-oriented' resolution of nested control structures. For instance, **for**$(x \leftarrow H)$ **for**$(x' \leftarrow H')\ Z$ is in normal form, but SQL's SELECT statement allows multiple tables as its input (implicit inner join), and so is our IR. We express it as a single ***select***-term which has $(x \leftarrow H)$ and $(x' \leftarrow H')$ as its input. The above

translation collects such $H$ and $H'$ by recursively traversing the term (the first clause for **for**).

The term $\mathbf{for}(x \leftarrow H)\ \mathcal{G}_{(\kappa,\alpha)}(U)$ is similarly translated (the second clause for **for**), but it needs one more twist. This term performs grouping for each $x$, namely, if $H$ has $N$ rows, grouping takes place $N$ times. To simulate the behavior by a single grouping operation, we add all the fields of $x$ to the grouping keys ($[\overline{x.l}]$ @ $K$) so that the grouping operation can distinguish records from different values of $x$. We can optimize the translation; if the table for $H$ has the field for primary key, which means that each record in the table can be distinguished by the value of the primary key, then we only have to add the primary key to the grouping keys.

In this way, all (output-oriented) nesting of **for** is resolved and we get a single ***select***-term or ***gselect***-term.

As an example, the term $Q$(**simple**) in Section 2 is translated to the following IR term:

$$\begin{aligned}&\boldsymbol{gselect}([(x \leftarrow \boldsymbol{sub}(\boldsymbol{select}([(o \leftarrow \boldsymbol{table}(\text{“orders”}))],\\ &\qquad\qquad o.*,\ o.\text{date} = \text{“}\texttt{2020-01-01}\text{”})))],\\ &\quad \{\text{pid} = x.\text{pid}, \text{qty\_sum} = \text{SUM}(x.\text{qty})\},\\ &\quad \mathbf{true},\ [x.\text{pid}],\ \mathbf{true})\end{aligned}$$

### 4.2 Reducing Query Size

We first transform the pattern $g \circ f$ where $g$ is an operation with grouping and $f$ is an operation without grouping. This combination is translated to the following IR term:

$$\begin{aligned}&\boldsymbol{gselect}([(y \leftarrow \boldsymbol{sub}(\boldsymbol{select}(I',\ O',\ C')))],\ O,\ C_1,\ K,\ C_2)\\ &\rightarrow\ \boldsymbol{gselect}(I', O\,[y := O'], C_1\,[y := O'] \wedge C',\\ &\qquad\qquad K\,[y := O'], C_2\,[y := O'])\end{aligned}$$

The rule can be explained as follows. The query in the left-hand side of this rule first executes the subquery $\boldsymbol{sub}(\cdots)$ to get a table as its value, then the outer query performs grouping on the table. We can express the above combination by a single statement in SQL. Note that the variable $y$ in the term before the transformation is bound in $O$, $C_1$, $K$, and $C_2$, hence we need to substitute $O'$ for $y$ in the result of transformation. Also the conditions $C'$ and $C_1$ (which corresponds to the pre-condition for grouping) are merged by conjunction.

Substitution for IR terms is in general dangerous, as the correspondence between the grouping operation and aggregate functions may be broken when a term that contains aggregate functions is substituted for a variable which is in the scope of a different grouping operator.[7] The above transformation rule does not suffer from this problem since the term $O'$ is a simple record and does not contain aggregate functions. We need to consider this problem seriously, if our language allows subqueries to appear in expressions such as the term $\boldsymbol{exists}(M)$, which is left for future work.

[7] This is one of the reasons why introduced a $\mathcal{G}$-operator which takes aggregate functions as its parameter in Quelg. Since they are coupled by our operator, we do not have to worry about the problem.

It is easy to show that the above transformation preserve typing, and thus we do not lose the important property of the absence of nested data structures.

The transformation reduces the depth of nested control structures in queries, yet, it is limited to the case when the input list of ***gselect*** is a singleton, and does not necessarily improve the size and performance of queries, which is highly dependent on the terms being transformed. See Section 7 for our experiments on these issues.

The above rule can be generalized in several ways. For instance, we can generalize it to the rule where the input list has multiple elements such that all the elements but the last one are the pairs of a variable and a table expressions. For this case, we only have to enumerate the table expressions after the FROM clause in SQL to obtain the equivalent query. More general cases are harder, which is left for future work.

The next rule transforms the pattern $f \circ g$ as follows:

$$\begin{aligned}&\boldsymbol{select}([(y \leftarrow \boldsymbol{sub}(\boldsymbol{gselect}(I,\ O',\ C_1,\ K,\ C_2)))],\ O,\ C)\\ &\rightarrow\ \boldsymbol{gselect}(I,\ O\,[y := O'],\ C_1,\ K,\ C\,[y := O'] \wedge C_2)\end{aligned}$$

This rule is very similar to the previous case. All the remarks after the first transformation applies to this rule, too.

As an example, the IR term for $Q$(**simple**) in Section 4.1 is transformed to the following term:

$$\begin{aligned}&\boldsymbol{gselect}([(o \leftarrow \boldsymbol{table}(\text{“orders”}))],\\ &\qquad \{\text{pid} = o.\text{pid}, \text{qty\_sum} = \text{SUM}(o.\text{qty})\},\\ &\qquad o.\text{date} = \text{“}\texttt{2020-01-01}\text{”},\\ &\qquad [o.\text{pid}],\ \mathbf{true})\end{aligned}$$

As expected, the size of the query has been reduced.

### 4.3 Eliminating Correlated Subquery

We consider the optimization for eliminating *correlated subqueries*. In our language, correlated subqueries are nested subqueries in which a variable bound by one subquery is dereferenced in the other subquery. It is well known that correlated subqueries often show rather poor performance. Although an experienced SQL writer would not write correlated subqueries unless it is absolutely necessary, our transformation-based approach may sometimes introduce correlated subqueries. Hence, we study this issue within our framework, and show that it is possible to eliminate correlated subqueries to some extent, which leads to a great improvement on the performance.

A query $\mathbf{for}(x \leftarrow \mathbf{table}(t))\ \mathbf{for}(y \leftarrow \mathcal{G}_{(\kappa,\alpha)}(M))\ N$ in Quelg is transformed to the following IR term:

$$\begin{aligned}&\boldsymbol{select}([(x \leftarrow \boldsymbol{table}(t));\\ &\qquad (y \leftarrow \boldsymbol{sub}(\boldsymbol{gselect}(I',\ O',\ C_1,\ K',\ C_2)))]),\ O,\ C)\end{aligned}$$

It is a correlated subquery if the variable $x$ is used in the ***gselect***-term. To optimize it while preserving the semantics, we need to combine the input of ***select*** into one subquery. This can be achieved by adding all the records in the table $t$ to the output ($O'$) and the grouping key ($K'$) of the subquery.

The translation can be expressed as follows:

$$
\begin{array}{l}
\textbf{\textit{select}}([(x \leftarrow \textbf{\textit{table}}(t));\\
\qquad (y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{gselect}}(I',\ O',\ C_1,\ K',\ C_2)))],\ O,\ C)\\
\rightarrow\ \textbf{\textit{select}}([(y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{gselect}}((x \leftarrow \textbf{\textit{table}}(t)) :: I',\\
\qquad\qquad \overline{\{l' = x.l\}} \ @\ O',\ C_1,\ [\overline{x.l}] \ @\ K',\ C_2)))],\\
\qquad O\,[\overline{x.l := y.l'}],\ C\,[\overline{x.l := y.l'}])
\end{array}
$$

This rule transforms a ***select***-term which has two inputs: the first one is a table term, and the second one is a grouping term. The whole term is a correlated subquery if $x$ occurs in the rest of the query. The operator @ for records denotes concatenation. We use filed names $l'$ in several places but for the moment let us ignore the difference between $l'$ and $l$.

The rule adds the table (***table***$(t)$) to the input of the grouping term. The occurrences of $x$ in $O$ and $C$ are substituted for $y$, thus dependency on $x$ is eliminated and the resulting query is no more a correlated subquery. We also add all the fields in $x$ to the key list of grouping to preserve the semantics of the query. Put differently, the query after the transformation takes the tagged (disjoint) union for each $x$, and the grouping operation is applied to the resulting bag, rather than iterating grouping operations for each $x$.

One subtle issue remains in this transformation, namely, the role of renaming field names from $l$ to $l'$. Its role is to avoid the possible collision of field names between $x$ and $y$. Namely, the field $x.l$ is added to $y$, but if $y$ also has the field $y.l$, then we copy the value of $x.l$ to $y.l'$. For simplicity, we rename all the fields in $x$ regardless they have collision with $y$. By this trick, we can successfully translate a correlated subquery to a non-correlated one while keeping semantics.

### 4.4 Summary of Our Transformation

Fig. 11 summarizes the optimization rules other than the translation to IR.

Actually the third optimization for eliminating correlated queries is generalized to the case where the first input is not necessarily a table expression, but an arbitrary expression (***select*** or ***gselect***).

We take the left-most strategy when we apply these optimization rules to IR terms. As we will see later, by composing the translations in Fig.10 with these optimizations we get smaller, and sometimes better performant, queries than those obtained by our previous work.

## 5 Translation to SQL

As the final step of our translations, we translate IR to SQL. For a technical reason explained below, we assume that the back-end SQL is PostgreSQL (or other dialects which have an equivalent feature).

Our IR is designed to be a compact macro language for SQL, so the translation between them should be identity module the names of constructors: for an IR term ***select*** $(I, O, C)$, we map $I$ to the FROM clause, $O$ to SELECT, and $C$ to WHERE.

**(Stage 1)**

$$
\begin{array}{l}
O_1[\![\textbf{\textit{gselect}}(I,\ O,\ C_1,\ K,\ C_2)]\!] \quad = \\
\quad \textbf{\textit{gselect}}(I', O\,[y := O'],\\
\qquad\qquad C_1\,[y := O'] \wedge C', K\,[y := O'], C_2\,[y := O'])\\
\qquad (\textit{where}\ I = [(y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{select}}(I',\ O',\ C')))])
\end{array}
$$

**(Stage 2)**

$$
\begin{array}{l}
O_2[\![\textbf{\textit{select}}(I,\ O,\ C)]\!] \quad = \\
\quad \textbf{\textit{gselect}}(I',\ O\,[y := O'],\ C_1,\ K',\ C\,[y := O'] \wedge C_2)\\
\quad (\textit{where}\ I = [(y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{gselect}}(I',\ O',\ C_1,\ K',\ C_2)))])
\end{array}
$$

**(Stage 3)**

$$
\begin{array}{l}
O_3[\![\textbf{\textit{select}}(I,\ O,\ C)]\!] \quad = \\
\qquad O_2[\![\textbf{\textit{select}}(I'',\ O\,[\overline{x.l := y.l'}],\ C\,[\overline{x.l := y.l'}])]\!]\\
\left(\begin{array}{l}
\textit{where query} = \textbf{\textit{table}}(t)\ \textit{or}\ \textbf{\textit{sub}}(\textbf{\textit{select}}\ \textit{or}\ \textbf{\textit{gselect}})\\
\textit{where}\ I = [(x \leftarrow \textit{query});\\
\qquad\qquad (y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{gselect}}(I',\ O',\ C_1,\ K',\ C_2)))]\\
\rightarrow\ I'' = [(y \leftarrow \textbf{\textit{sub}}(\textbf{\textit{gselect}}((x \leftarrow \textit{query}) :: I',\\
\qquad\qquad\qquad \overline{\{l' = x.l\}} \ @\ O', C_1, [\overline{x.l}] \ @\ K', C_2)))]
\end{array}\right)
\end{array}
$$

**Figure 11.** Optimizations

However the naive translation has a subtle problem as follows. As we explained in Section 4.1, if $I = [(x_1 \leftarrow T_1); (x_2 \leftarrow T_2)]$, then $T_2$ may dereference $x_1$, namely, the input tables may have dependency. As the input tables (the expressions after FROM) in SQL does not allow dependency, the naive translation fails. In fact, the following query in SQL raises an error when executed.

```
SELECT *
FROM products AS p,
     (SELECT * FROM orders AS o
               WHERE p.pid = o.pid) AS x
(* ERROR: invalid reference to FROM-clause
    entry for table "p" *)
```

We solve this problem using the LATERAL clause provided by recent PostgreSQL,[8] which makes it possible to allow dependency among input tables. With the LATERAL clause, the above query is written as follows.

```
SELECT *
FROM products AS p,
     LATERAL (SELECT *
              FROM orders AS o
              WHERE p.pid = o.pid) AS x
```

Executing this query has no problem.

Fig. 12 defines the translation from IR to SQL (PostgreSQL). We assume that bound variables in IR terms are mutually

[8]The feature has been supported by PostgreSQL since 9.3, and by MySQL since 8.0.14.

$$
\begin{aligned}
\mathcal{S}[\![\textbf{\textit{unionall}}(U_1, U_2)]\!] &= \mathcal{S}[\![U_1]\!]\ \textbf{UNION ALL}\ \mathcal{S}[\![U_2]\!] \\
\mathcal{S}[\![\textbf{\textit{select}}(I, O, C)]\!] &= \textbf{SELECT}\ \mathcal{S}[\![O]\!]\ \textbf{FROM}\ \mathcal{S}[\![I]\!]\ \textbf{WHERE}\ \mathcal{S}[\![C]\!] \\
\mathcal{S}[\![\textbf{\textit{gselect}}(I, O, C_1, K, C_2)]\!] &= \textbf{SELECT}\ \mathcal{S}[\![O]\!]\ \textbf{FROM}\ \mathcal{S}[\![I]\!]\ \textbf{WHERE}\ \mathcal{S}[\![C_1]\!]\ \textbf{GROUP BY}\ \mathcal{S}[\![K]\!]\ \textbf{HAVING}\ \mathcal{S}[\![C_2]\!] \\
\mathcal{S}[\![\textbf{\textit{table}}(t)]\!] &= t \\
\mathcal{S}[\![\textbf{\textit{sub}}(U)]\!] &= \textbf{LATERAL}\ (\mathcal{S}[\![U]\!]) \\
\mathcal{S}[\![c]\!] &= c \\
\mathcal{S}[\![x.l]\!] &= x.l \\
\mathcal{S}[\![\oplus(\overline{B})]\!] &= \oplus_{sql}(\overline{\mathcal{S}[\![B]\!]}) \\
\mathcal{S}[\![[(x \leftarrow T)]]\!] &= \overline{\mathcal{S}[\![T]\!]\ \textbf{AS}\ x} \\
\mathcal{S}[\![\{\overline{l = B}\}]\!] &= \overline{\mathcal{S}[\![B]\!]\ \textbf{AS}\ l} \\
\mathcal{S}[\![x]\!] &= x.* \\
\mathcal{S}[\![C \wedge C']\!] &= \mathcal{S}[\![C]\!]\ \textbf{AND}\ \mathcal{S}[\![C']\!] \\
\mathcal{S}[\![[\overline{B}]]\!] &= \overline{\mathcal{S}[\![B]\!]}
\end{aligned}
$$

**Figure 12.** Translation from IR to SQL

distinct, and the same variable names can be used in the corresponding SQL.

The above translation works for all the terms in our IR, which completes the task for transforming a language-integrated query to a single executable SQL query.

## 6 Involved Examples

In this section, we apply our transformations to several examples with grouping and aggregate functions. They use the tables in Fig.3 in Section 2. We assume that examples use our grouping operator from the first place, hence we do not need to decompose them.

### 6.1 Query with Nested Data Structure

The first query $Q$(**getSales**) uses a nested data structure and grouping, shown below.

$$
\begin{aligned}
&Q(\textbf{getSales}) = \\
&\quad \mathcal{G}_{(\text{pid},\alpha)}(\textbf{for}(x \leftarrow \textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
&\qquad\qquad \textbf{yield}\ \{\text{order} = \textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
&\qquad\qquad\qquad\qquad \textbf{where}\ (p.\text{pid} = o.\text{pid}) \\
&\qquad\qquad\qquad\qquad \textbf{yield}\ \{\text{pid} = o.\text{pid}, \\
&\qquad\qquad\qquad\qquad\ \ \text{sales} = p.\text{price} * o.\text{qty}\}\}) \\
&\qquad \textbf{for}(y \leftarrow x.\text{order}) \\
&\qquad \textbf{yield}\ y) \\
&\quad \text{where}\ \alpha\ =\ \{(\text{sales}, \text{AVG}, \text{sales\_avg})\}
\end{aligned}
$$

$Q$(**getSales**) fetches the record with the same pid from the products table and orders table and calculates the sales. It performs grouping based on pid as the key, and calculates the average of sales. It uses a nested data structure. The query is transformed to the following term.

$$
\begin{aligned}
Q(\textbf{getSales}) = \mathcal{G}_{(\text{pid},\alpha)}(&\textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
&\textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
&\textbf{where}\ (p.\text{pid} = o.\text{pid}) \\
&\textbf{yield}\ \{\text{pid} = o.\text{pid}, \\
&\qquad \text{sales} = p.\text{price} * o.\text{qty}\}) \\
\text{where}\ \alpha\ =\ &\{(\text{sales}, \text{AVG}, \text{sales\_avg})\}
\end{aligned}
$$

We can optimize $Q$(**getSales**) by using only the Stage 1 optimization, and get the following IR term.

$$
\begin{aligned}
&\textbf{\textit{gselect}}([(p \leftarrow \textbf{\textit{table}}(\text{“products”})); (o \leftarrow \textbf{\textit{table}}(\text{“orders”}))], \\
&\quad \{\text{pid} = o.\text{pid}, \text{sales\_avg} = \text{AVG}(p.\text{price} * o.\text{qty})\}, \\
&\quad p.\text{pid} = o.\text{pid},\ \textbf{true},\ [o.\text{pid}],\ \textbf{true})
\end{aligned}
$$

By translating this IR term, we get the following compact SQL query:

```
SELECT o.pid AS pid,
       AVG(p.price * o.qty) AS sales_avg
FROM products AS p, orders AS o
WHERE p.pid = o.pid
GROUP BY o.pid
HAVING TRUE
```

### 6.2 Query with Correlated Subquery

The second query $Q$(**getQty**) uses a correlated subquery, shown below:

$$
\begin{aligned}
&Q(\textbf{getQty}) = \\
&\quad \textbf{for}(p \leftarrow \textbf{table}(\text{“products”})) \\
&\quad \textbf{for}(y \leftarrow \mathcal{G}_{(\{\text{name},\text{date}\},\alpha)}(\textbf{for}(o \leftarrow \textbf{table}(\text{“orders”})) \\
&\qquad\qquad\qquad\qquad \textbf{where}\ (p.\text{pid} = o.\text{pid}) \\
&\qquad\qquad\qquad\qquad \textbf{yield}\ \{\text{name} = p.\text{name}, \\
&\qquad\qquad\qquad\qquad\qquad\ \ \text{date} = o.\text{date}, \\
&\qquad\qquad\qquad\qquad\qquad\ \ \text{qty} = o.\text{qty}\})) \\
&\qquad \textbf{yield}\ \{\text{name} = p.\text{name}, \text{date} = y.\text{date}, \\
&\qquad\qquad \text{qty\_sum} = y.\text{qty\_sum}\} \\
&\quad \text{where}\ \alpha\ =\ \{(\text{qty}, \text{SUM}, \text{qty\_sum})\}
\end{aligned}
$$

The query $Q$(**getQty**) retrieves records with the same pid from the products and orders tables, then performs grouping by the name and dates fields as keys. It calculates the total number of the products for each date. It is already in normal form, and we translate it using all optimizations from Stages 1 through 3. The IR term obtained by applying the Stage 1

optimization only is shown below:

```
select([(p ← table("products"));
        (y ← sub(gselect([(o ← table("orders"))],
                         {name = p.name, date = o.date,
                          qty_sum = SUM(o.qty)},
                         p.pid = o.pid,
                         [p.name; o.date], true)))],
       {name = p.name, date = y.date,
        qty_sum = y.qty_sum},
       true)
```

The IR term obtained by applying the Stage 2 and 3 optimizations for this term is shown as follows:

```
gselect([(p ← table("products")); (o ← table("orders"))],
        {name = p.name, date = o.date,
         qty_sum = SUM(o.qty)},
        p.pid = o.pid,
        [p.pid; p.name; p.cat; p.price; o.date], true)
```

Finally, we translate it to SQL, obtaining the following query.

```
SELECT p.name AS name, o.date AS date,
       SUM(o.qty) AS qty_sum
FROM products AS p, orders AS o
WHERE p.pid = o.pid
GROUP BY p.pid, p.name, p.cat, p.price,
         o.date
HAVING TRUE
```

Note that we have obtained a single query without subqueries. If we had not used the optimizations in this paper, we would have got a SQL query with a correlated subquery, which would be rather inefficient.

## 7 Performance

We have implemented our transformations in OCaml based on Suzuki et al.'s typed tagless-final implementation for their language-integrated query [12], and compared the performance by measuring the execution time of SQL queries generated by our previous work and by the present paper. We also measured the execution time of SQL queries generated by Microsoft's LINQ in F#. The computing environment is Mac OS 10.13.2 with Intel Core i5-7360 CPU with 8GB RAM, and we use OCaml 4.07.1, F# 4.7 using .NET Core 3.1, and PostgreSQL 11.5. All queries used in the experiment are available online.[9]

We have used the tables in Fig.3 where the number of rows is increased to 5000 (products) and 10000 (orders), and also used several other tables. We have measured the performance for eleven queries with grouping, including the one in Section 6. The queries used in this experiments are classified into two groups.

[9] http://logic.cs.tsukuba.ac.jp/~rui/quelg_opt/example.html

**Table 1.** Results of Performance Measurement

| Query | LINQ | Quelg | Quelg-opt | Trans |
|---|---|---|---|---|
| $Q$(**simple**) | 3.18 | 3.57 | 3.18 | 0.03 |
| $Q(g_2 \circ f \circ g_1)$ | Av | 6.26 | 6.99 | 0.05 |
| $Q$(**getSales**) | Av | 4.38 | 4.35 | 0.22 |
| $Q$(**getCount**) | 2.75 | 3.60 | 2.75 | 0.03 |
| $Q$(**abstraction**) | 2.14 | 2.51 | 2.14 | 0.06 |
| $Q$(**predicate**) | NA | 1.42 | 0.99 | 0.03 |
| $Q$(**getScore**) | NA | 29.56 | 25.61 | 0.36 |
| $Q$(**getQty**) | NA | 1855.55 | 15.91 | 0.06 |
| $Q$(**multiple**) | NA | 19275.64 | 16.10 | 0.04 |
| $Q$(**for**-$\mathcal{G}$) | NA | 26.14 | 22.59 | 0.04 |
| $Q$(**compose**) | NA | 3.17 | 1.52 | 0.14 |

Time unit is millisecond.

The first group consists of relatively simple queries which do not generate correlated subqueries even if we naively generate SQL queries (hence we do not need the Stage 3 optimization for these queries). We have seven queries in this group (the upper half of Table 1). $Q$(**simple**) and $Q$(**getSales**) are the ones in Sections 2.1 and 6.1, respectively, and $Q(g_2 \circ f \circ g_1)$ is an instance of the pattern $g_2 \circ f \circ g_1$ in Section 2.2. $Q$(**getCount**) performs grouping twice by nested $\mathcal{G}$-operators, $Q$(**abstraction**) combines lambda abstraction and a $\mathcal{G}$-operator, $Q$(**predicate**) groups the data using the predicate given as the argument of the query, and $Q$(**getScore**) used nested data structures as intermediate data in the computation with grouping.

The second group consists of queries that handle correlated subqueries, which need all of Stage 1 through 3 optimizations to optimize. There are four queries in this group (the lower half of Table 1). $Q$(**getQty**) is the one in Section 6.1, $Q$(**multiple**) is a query in which a table and the $\mathcal{G}$-operator are mixed in the input of each **for**-constructor, $Q$(**for**-$\mathcal{G}$) has the $\mathcal{G}$-operator in the output of the **for**-constructor and is translated to a SQL query with correlated subqueries, and $Q$(**compose**) combines two queries with grouping.

Table 1 shows the execution time of SQL queries for three cases: LINQ (the first column), Quelg without optimization (the second), and Quelg with optimization (the third). The last column (Trans) shows the time for transforming queries by the method in the present paper. The time unit is millisecond. In the table, NA (not available) means the query cannot be executed, and Av (avalanche) means that the query caused the query avalanche ($N$ + 1-query) problem.

Note that the time for transforming queries is much shorter than the execution time, and is negligible. This is as expected, as our transformation rules are relatively simple as program transformations. This is a preferable result, since we may have to generate SQL queries dynamically in language-integrated query (while executing our high-level code in F#

or other programming languages), and the time for code generation matters for such cases.

We compare Microsoft's LINQ and our methods. LINQ successfully generated and executed three queries only. There are two reasons for this. First, LINQ does not transform queries with grouping, hence a query with grouping which has nested control structures causes the $N + 1$-query problem. In this experiment, the sizes of input tables are 500 or 1000, hence approximately 500 or 1000 SQL queries are generated and sent to SQL processors, which needs a huge amount of time (results marked with Av). The second reason is that LINQ does not support the LATERAL clause, and queries with correlated subqueries cannot be generated (results marked with NA).

On the other hand, simple queries such as $Q$(**simple**), $Q$(**getCount**), and $Q$(**abstraction**) run as fast as our optimized one, since they do not need normalization or the LATERAL clause. Our un-optimized result (the second column) sometimes performs worse than LINQ, since we decomposed a single query, which increases the complexity of queries, and if no normalization is needed, the decomposition (without optimization) has a bad effect on performance.

We then compare the performance of our method with and without optimizations. For the queries from $Q$(**simple**) to $Q$(**getScore**) (simpler queries), our optimization does not have a big impact on the execution time even though our optimization succeeded in reducing the size of generate queries. This can be explained by the fact that the query optimizer in PostgreSQL improves the performance of redundant queries before our optimization.

For the queries $Q$(**getQty**) and $Q$(**multiple**), which use grouping, nested control structures, and generate correlated subqueries, our optimization has shown great improvements on performance with the ratio from 15% to 1097%. The ratio varies from one example to another, however, a similar improvement has been observed as long as we have tested. These results are in favor of our work.

Optimization for correlated subqueries is a big issue in database studies, which is beyond the scope of this paper, but we think that our program-transformation (and program-generation) approach is beneficial to study the issue when the target language contains grouping, as we can make use of the standard technique for program transformations.

Finally we mention the size of generated queries which is not on the table. For all cases, the size of generated queries was reduced by our optimization from 33% to 67%, where we measured the size of queries by the number of SELECT statements in the query. Query size does not matter for small queries, but it sometimes matters since many database engines have the upper limit on the number of subqueries or of query size, and the program-transformation approach may occasionally generate large queries. Hence, the smaller size of queries is preferable even if the performance remains the same.

The experiments we have conducted use relatively small queries, and thorough experiments using larger examples from practical applications are left for future work.

## 8 Conclusion

Language-integrated query reduces the impedance mismatch problem, and provides high-level abstractions to programmers when they write database queries. Cooper's influential paper has the title "The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed" which suggests us that writing a SQL query in your own, favorite programming language is a dream of script-writers. The "dream" does not come true without cost; despite many researchers' efforts, transforming a query with grouping and aggregation into an efficient SQL query has been an open problem for years. In our previous work, we solved this problem when the target SQL allows nested control structures (subqueries), but their solution has another problem of excessively large queries and their poor performance.

This paper solves the problem by re-organizing the query to obtain a smaller query with fewer control structures. The result of our experiments is encouraging; SQL queries generated by our method are smaller and outperforms those generated by Microsoft's LINQ, and one by our previous work.

Research on language-integrated query is still a hot topic. For example, Kiselyov et al. has proposed a technique to cover the ORDER BY clause in language-integrated query [7]. Ricciotti et al. proved strong normalization for both homogeneous and heterogeneous queries [11].

Let us state future work other than those already mentioned in this paper.

The next step of the present work is to give an efficient implementation of the program transformation. If we generate SQL queries only statically, the time for the transformation does not matter, however, when we dynamically generate SQL queries in language-integrated query the transformation time matters. Formal verification of our transformation for suitable semantics is also an important remaining work.

From the practical point of view, extending the source language Quelg to cover a larger subset of SQL than the present one is an interesting topic. In this work, we allow subqueries to appear in the FROM or SELECT clause, but subqueries must not appear in, for instance, the WHERE clause in this study. Classic database theory investigated such subqueries, and we hope to combine our results with them as future work.

Acknowledgements. We would like to thank anonymous reviewers for constructive comments. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) No. 18H03218 and No. 17H01724A.

## References


[1] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 403–416. https://doi.org/10.1145/2500365.2500586

[2] James Cheney, Sam Lindley, and Philip Wadler. 2014. *Language-integrated query using comprehension syntax: state of the art, open problems, and work in progress*. Technical Report. http://popl.mpi-sws.org/2014/dcp2014/cheney.pdf, (Accessed on Oct 2020).

[3] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *Database Programming Languages - DBPL 2009, 12th International Symposium, Lyon, France, August 24, 2009. Proceedings*. 36–51. https://doi.org/10.1007/978-3-642-03793-1_3

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.

[5] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY: Database-Supported Program Execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 1063–1066. https://doi.org/10.1145/1559845.1559982

[6] Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. *PVLDB* 3, 1 (2010), 162–172. https://doi.org/10.14778/1920841.1920866

[7] Oleg Kiselyov and Tatsuya Katsushima. 2017. Sound and Efficient Language-Integrated Query - Maintaining the ORDER. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. 364–383. https://doi.org/10.1007/978-3-319-71237-6_18

[8] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. 706. https://doi.org/10.1145/1142473.1142552

[9] Atsushi Ohori and Katsuhiro Ueno. 2011. Making Standard ML a Practical Database Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 307–319. https://doi.org/10.1145/2034773.2034815

[10] Rui Okura and Yukiyoshi Kameyama. 2020. Language-Integrated Query with Nested Data Structures and Grouping. In *Functional and Logic Programming*, Keisuke Nakano and Konstantinos Sagonas (Eds.). Springer International Publishing, Cham, 139–158.

[11] Wilmer Ricciotti and James Cheney. 2020. Strongly Normalizing Higher-Order Relational Queries. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:22. https://doi.org/10.4230/LIPIcs.FSCD.2020.28

[12] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 37–48. https://doi.org/10.1145/2847538.2847542

[13] Val Tannen, Peter Buneman, and Limsoon Wong. 1992. Naturally Embedded Query Languages. In *Database Theory - ICDT'92, 4th International Conference, Berlin, Germany, October 14-16, 1992, Proceedings*. 140–154. https://doi.org/10.1007/3-540-56039-4_38