A novel framework for synthesizing nested queries in SQL from business requirements language

Mathew George¹, Dr. Rohini V²

¹,²Department of Computer Science CHRIST, Bangalore

Article History: Received: 11 January 2021; Accepted: 27 February 2021; Published online: 5 April 2021

Abstract: Different methods and systems were proposed in the past for translating Natural Language (NL) statements in to Structured Query Language (SQL) queries. Translating statements resulting in 'nested' queries have always been a challenge and was not effectively handled. This work proposes a framework for translating requirement statements resulting in the construction of nested Queries. While translating nested scenarios; often there is a need to create sub-queries that execute in pipeline or in parallel or both operating together. Lambda Calculus is found to be effective in representing the intermediate expressions and helps in performing the transformations that are needed in translating specific predicates into SQL, but its inflexibility in combining parallel computations is a constraint. To represent clauses that are in parallel or arein pipeline, and to perform the required transformations on the intermediate expressions involving these, more advanced programming constructs are needed. This work recommends the use of advanced language constructs and adoptions of functional programming techniques for performing the required transformation at the intermediate language level.

Keywords: Bags, Combinator, Initial Algebra, Orthogonal, Structural Recursion, Monad Comprehensions, Folds.

1. Introduction

Most of the earlier efforts in automating SQL creation from Data Requirement Statements were in the form of Rewrite systems that provided a platform for intermediate representation and offered an astandard method for modeling computation [4]. The choice of an adequate intermediate representation is a major step in the overall translation and repair process. For creating nested SQLs we need an intermediate programming paradigm that has the semantic simplicity of relational algebra, and the expressive power of functional programming languages. Hence special emphasis is kept on the intermediate language representation and the application of required transformation techniques in the ambit of a complete translation framework. The type-based design based on initial algebras³ of a core functional language is followed and intermediate representationsthat suits the demands of nested query generation is subsequently developed. Advanced type systems are needed in the design of an intermediate language for representing nested queries. Applying relational query processing rules alone will not be sufficient to represent and to perform transformations on these extended type systems.

For generating nested queries, it is imperative to define a calculus and a language that can represent comprehension syntax and perform the required operations as relational calculus does to relational query languages. Its main processing requirement is to perform structural recursion⁷ on bulk data types like bags³ and sets. This intermediate programming paradigm should also be able to perform recursion on bags of data traversing through different levels of a tree structure. The difference with regular functional programming languages is that this language is built around a restricted form of structural recursion.

In the NL to SQL translation domain, comprehensions⁸ and basic Combinators⁷together in effect can represent and meet the transformation requirements of the intermediate language. The comprehension calculus provides the means to canonically represent and effectively reason about complex predicates, including quantifiers, and collection processing. Advanced programming constructs like Monad Comprehensions⁹ and Folds¹⁰ can significantly ease our efforts in combining and translating nested clauses that gets attached to the main SQL trunk. In fact, Monad Comprehensions and Folds are implementations of structural recursion. The main argument of this article is that with the help of advanced type systems and the application of functional programming techniques can provide the adequate framework for the automatic derivation of SQLs from Data Requirement Specifications.

The organization of the paper is as follows: A Novel framework to The next section discusses the ‘RELATED WORK’ and the progress made by research communities in synthesizing nested queries. The key technical contributions and their inclusion in the transformation framework are given in the OVERVIEW Section. Detailed concepts and their relevance followed by theoretical walk through can be seen in the section on CURRENT WORK. A motivating example and the steps for translating a sample nested scenario, is given in the section named ‘A CASE STUDY’. The SCOPE
FOR FUTURE WORK and the CONCLUSIONS are described in the last two sections. A BIBLIOGRAPHY of terms and the details of literature referred can be found in the REFERENCES section.

2. Related work

Yaghmazadeh N. and Dillig I. (2017) proposed a type and database content driven synthesize-repair framework [1] for synthesizing SQLs from Natural language statements. Rewrite methods and Inference Rules based transformations are central to their work. The method suggested for synthesizing nested queries is to repeat the same process used for generating the main query. The pipelining and dynamic re-organization required while combining sub-queries cannot be brought out easily through rule based rewriting techniques or by simply repeating the process used for generating the main query. In 2018, Hosu et al. proposed a sketch-based two-step neural encoder model [12] known as SEQ2SEQ for generating SQLs based on a user’s requirement specification in natural language. But this needs to be extended for complex cases involving nested queries where operations based on nested structures are inevitable. Grust T. and Scholl, M. H. (1999) suggested a type-based core functional language based on initial algebras as intermediate representation which can be transformed by applying advanced functional programming techniques.

Earlier, algebraic approaches dominated the intermediate language representation of query structures used in translation. SQL is predominantly designed from abstractions given by relational algebra. A key observation is that relational algebra operates on sets while SQL is primarily based on bags [2] and the query algebra operators are in fact abstract representation of underlying procedures implemented by the query engine. Query predicates were viewed as annotations to algebraic operators and were not part of the translation or compilation phase but were treated later during optimization phase. Hence adopting functional programming techniques becomes a necessity to bring the flexibility and composition required for creating and integrating nested queries. Imparting functional outlook to automatic creation of SQLs makes it disposed for an extensive collection of program transformation techniques in the category of Bird-Meetan’s [7] formalism.

For translating SQLs into Object Code and to perform subsequent optimization, Grust T and Scholl M. H. (1999) [2] describes an intermediate language based on Combinators, extended further by applying functional programming techniques like structural recursion and subsequently implemented through comprehensions. Their work deals with Query compilation and optimization and not on Query Synthesis from Natural Language. But the techniques discussed are relevant and can be adapted for SQL synthesis. Hence adopting functional programming techniques becomes a necessity to bring the flexibility and composition required for creating and integrating nested queries.

3. Overview

In the NL to SQL translation scenario, translating relational algebraic expressions into SQL can be impaired by the type system mismatches between them. This discrepancy between the intermediate representation and query languagemakes the translation complex. Hence it is mandatory to bring higher-order functional programming techniques invented by the functional programming communities at the intermediate language level to deal with this impedance mismatch. As the intermediate language is functional in nature, functional programming techniques can be applied to the expressions and components of the intermediate representation to transform it to produce the desired structured Query representation. Functional abstraction at the intermediate language level facilitates refactoring of query fragments into parameterized functions, enables the formation of nested intermediate data structures for which no relational algebra equivalent be easily drawn.

This work follows the type-based design of the intermediate language than the operation-based design where query operators greatly influence the design. At the core of the functional programming is the capability to introduce new datatypes and to define functions that manipulate their values. Referential transparency is an essential characteristic needed in transformational programming and equational reasoning. This is particularly important in the DRS to SQL translation as every relation can be defined as a type of the variables involved, though they belong to different contexts and often need to operate on the same equational plane. Combinators can be used to preserve the type and context of expressions. As long as typing rules are adhered, Combinators may be freely combined to make expressions that represent nested clauses.
Key Concepts
The key transformation techniques discussed in this paper include:
1. How Structural Recursion and its implementation in the form of Comprehensions can be effectively used to represent and implement sub-queries as a pipeline.
2. A more advanced functional programming construct-Monad Comprehension is proposed for function abstraction and for rallying expressions in a pipeline while translating from the Abstract Query Language into an equivalent SQL query.
3. Application of higher-order Combinators like folds (fold$^\text{r}$) for abstract representation and as a means to augment recursive processing initiated through Fixed-point Combinators.$^6$

System Architecture: This work takes over once an initial query sketch is generated (after relevant entities were identified from DRS using semantic parsing and passed as input parameters to the synthesis program), which needs to be repaired and extended further by employing the techniques described in this paper. The tool Rex (the query synthesis program) introduced in our earlier work$^1$ is extended further by implementing the advanced techniques discussed here.

![System Architecture](image)

**Figure 1: System Architecture**

4. Current work

Road map:
The components of the core intermediate language is defined first, followed by the transformations need to be performed on the expressions created using this language for achieving the translation. The entire work is centered on the application of functional programming techniques and scaling further by using higher-order functions and their associated operations as when needed. The pipelining techniques for effectively combining nested expressions are described next. The benefit of using Monad Comprehensions and fold$^\text{r}$s and their effectiveness in chaining and pipelining different translation components are key to the implementation of the concepts proposed and applied subsequently. Finally, a case study is presented to evaluate the effectiveness of the translation.

Significance of Structural Recursion
Recursion is the usual programming idiom for repeated execution on potentially infinite data tending towards termination on finite state machines. Structural recursion is a restricted form of recursion, that is declarative in nature and the form of the program follows the structure of the data$^5$. Programs written with structural recursion using a finite set of objects made from dynamic data types has the expressiveness of the relational algebra and can even scale up$^3$. Structural recursion makes it possible to express the requirements of iteration, aggregation, and quantification which are at the core of any database query language and is suitable for relatively complex nested SQLs creation scenarios.

Structural recursion is defined as a top-down, recursive function, much like tree traversing which evaluates the data top-down. Structural recursion is found in almost all the tree traversals. A desirable property for query languages is to restrict recursion of unordered regular trees to preserve their finiteness property. In contrast to general recursion, structural recursion always terminates. Structural recursion
can be organized into two identical ways but working in different directions, a) as a recursive function for data organized in different levels of a tree without revisiting traversed nodes to avoid infinite loops (that caters to multilevel nesting), b) as a bulk evaluation which processes the entire data in parallel using relational algebra operators for building parallel sub-queries that are attached to the main SQL trunk in their respective levels[11]. Nested queries have a natural correspondence to structural recursion. The DRS to SQL transformation program should take bags of input data, process it, move it out of bags by placing them into sets, by managing a flexible type system across while traversing different levels of the tree. Structural recursion allows the implementation of better algorithms for the same functionality that can be achieved through other programming techniques found around first class functions.

**Lambda Calculus combined with Comprehensions**

Functional languages are usually based on lambda calculus and supported by a solid equational theory that are eventually compiled and interpreted. A complete functional language is not needed to represent queries at an intermediate language level; instead, a small set of Combinators would suffice. Theoretically, complex queries can be formed from functional composition of higher order Combinators. Even though Combinators would suffice to represent closed predicates, the sublanguage with Combinators needs to be extended with functional programming techniques like comprehensions to combine and pipeline different components into a sequel. Also, while executing, interim results need to be communicated across operators because the functions implementing them are functioned to take their own specific inputs and passes intermediate results.

The Combinator sub-language, can be extend by applying the syntactic sugar - comprehensions which provides better abstraction of the query intermediate representation. As DRS is more declarative in nature than imperative, adopting Comprehensions have proven a very convenient construct in the creation of SQL kind of declarative query statements [12, 13]. As Comprehensions create data structures from iterators and combines loops and conditional tests in a compact way, they can be employed as an effective intermediary construct while translating DRS to SQL. Just like query languages, comprehensions are provided with variables, variable bindings and allows nesting of predicates arbitrarily without propagating side effects of any predicates involved. Hence comprehensions and basic Combinators typically complete the intermediate language [2].

Combinators may be orthogonally combined and freely rearranged as they are independent of each other due to their very nature. Combinators can be combined across query operators as well, since there are no interdependencies between operators. Internally, the Combinators are implemented with the help of indices. However, at execution time, Combinator algebra exposes its own limitation: especially when temporary results are communicated between operators and since these are designed separately to consume their own inputs, produce their intermediate results bringing out the necessity for meticulous combining to produce a resultant output. Comprehensions come to the rescue in such situations. Comprehensions connect related predicates with ease and are predisposed for query predicate transformations. Without the use of comprehensions this would have needed application of complicated sets of rewriting rules.

**Monad Comprehensions**

Monads provide a framework for bundling / structuring the semantic representation of features such as state, exceptions and continuations[8]. Monad Comprehensions are recommended for use at the intermediate language level to bundle related components and features, by arraying them in a pipeline to ensure connectivity and continuity between constituent parts. Different types of query nesting correspond to nested representations of Monad Comprehensions.

While Combinators facilitate abstraction of query operators and predicates, Monad comprehensions facilitate a calculus-style intermediate language. Calculus sub-expressions with the appropriate Combinators are similar to relational calculus but have better expressiveness. Apart from providing the needed syntactic sugaring, Monad Comprehensions provide all the benefits of a calculus-based query representation [10]. Moreover, due to its functional nature, program transformation techniques developed by the functional programming and the related data model communities can be applied on this intermediate language[2]. The type-based foundation and uniform representation of our intermediate language (IL) allows us to adopt functions (over values of an initial algebraic data type τ),
and structural recursion constructs like foldr provides the fundamental way to combine SQL predicates.

An intermediate language could also benefit much from higher-order function abstraction techniques like foldr in recombining the outputs of recursively processing constituent parts, by consistently replacing the structural components of a data structure with functions and values to construct a return value eventually. Foldr enables the implementation of the algebraic data type constructors as well as structural recursion as a single programming unit.

A typical form of a generic fold function is:

\[ \text{fold } f \ z \ \text{xs} \]

where:
- \( f \) is a higher-order function taking two arguments, an accumulator and an element of the list \( \text{xs} \). It is applied recursively to each element of \( \text{xs} \).
- \( z \) is the initial value of the accumulator and an argument of the function \( f \).
- \( \text{xs} \) is a collection (in fact queries map between the constructors of different collection types).

**Figure 2: Representation of foldr**

```
\[ e_1 \rightarrow f \ e_2 \rightarrow f \ e_3 \rightarrow z \]
```

**Figure 3: Foldr implementation**

Here, \( e_1, e_2, e_3 \) represents sub-query expressions and \([\text{]}\) represents a list object.

The head expression \( e(\text{select-from part}) \) of an SQL statement is defined as:

- \( e \rightarrow \) v (variables) | c (constants)
- p|q = predicate; f = aggregate function; s = subquery; t = term
- xs = table1; ys = table2; zs = table3 | collection
- x = field1; y = field2
- [ ] = Unknown table | Empty List
- \( f^t \) | (f) = Constructors
- \( \sigma \) = selection
- \( \tau \) = algebraic data type over a relation ranged by the below expression. Let \( l \) range over a set of labels

\[ \tau ::= \text{list} | \text{set} | \text{bag} | \text{unit} | \text{int} | \text{bool} | \text{string} | \text{real} | \tau \rightarrow \tau \]
A novel framework for synthesizing nested queries in SQL from business requirements language

The domain of a type $\tau$ itself can be deemed an algebra.

**Figure 4: Intermediate Language Definition**

The `select-from-where` block can be represented in the intermediate query language. The intermediate mapping construct $Q$ is represented as:

$$Q(\text{select } e \text{ from } e_1 \text{ as } x_1, \ldots, e_n \text{ as } x_n \text{ where } p) = [Q e | x_1 \leftarrow Q e_1 \ldots e_n \leftarrow Q e_n Q p]$$

A query clause $e$ may be compiled independently from sub-queries $e_i$ occurring in it. During the translation of $e$ the $e_i$ are treated as free variables that may be instantiated later to complete the translation.

In the comprehension $[e | q_1, \ldots, q_n]_\tau$ the predicates $q_i$ are either generators $v \leftarrow q_i$ or filters (expressions resulting in type bool). A generator $q_i = v \leftarrow q_i$ sequentially binds variable $v$ to elements of its range $q_i$. $v$ is bound in $q_{i+1}, \ldots, q_n$ and $e$. The binding of $v$ is propagated until a filter evaluates to False under the binding. The result of evaluating $e$ is collected in the list construction $\tau$. $\exists$.

**Figure 5: Algebraic Combinators (Monad Comprehension-based definitions)**

5. **A case study**

Prepare the SQL for the following DRS statement: “Retrieve `leave` details of all `employees` based on their `latest employment records`”.

**Fig6: DB Table Relationship**

Functional programs are constructed by knitting smaller programs together, using an intermediate list to communicate between the constituent parts. Lists are often used to glue separate components of a program together [9]. The key finding here is that Combinator based query predicates operate pretty much with listful programs. A listful program expresses a complex list manipulation by composition of generic Combinators, each generating an intermediate result list, which needs subsequent filtering.

Relational calculus can be deemed as a specialization of the Monad Comprehension calculus restricted over sets. The comprehension $[x | x \leftarrow xs, p x]$ is similar to the relational selection $\sigma_{p, xs}$ but more generalized to represent any data type of $\tau$ and can be shown as:
The nested comprehension is represented as:

\[
[f \mid x \in xs, \mu y \mid y \in ys, py^{distinct}]
\]

Translates to:

select distinct f x
from xs as x
where g x in (select h y
from ys as y
where p y)

Implemented as:

select distinct e.employee_id
from Employees e
where e.id in (select e.id
from Leave_Details
where a.employee_id = e.id);

As translation scheme can translate Monad Comprehensions into nested foldr expressions, which on executing the queries constructed from these expressions, the query engine will tag for nested-loop processing and executes. During the SQL creation process the sub-queries can be treated as free variables that may be progressively instantiated and inserted or appended to complete the SQL generation.

Fig 7: Intermediate representation of Employee-LeaveDetails for the latest employee_record in the database

Monad comprehensions and Combinators, the two different forms of syntactic sugar, put together to the basic recursion Combinator foldr, established connecting links in the intermediate representation for nested queries to be picked by the target query build program to subsequently create the needed SQL.
**Initial Sketch generation:**

To provide an example of nested queries, suppose that a user wants to retrieve the latest employment record based on leave details. We can express this query as:

$$\Pi_{\text{Emp_id, Name, emp_rcd, Designation, leave_code, from_date}} (\text{Leave_Details}) (\sigma_{e_id = \Pi_{\text{max(e_id, (Employees)}}) \land \sigma_{emp_rcd = \Pi_{\text{max(emp_rcd, (Employees)}})}$$

It is relatively easy to map the intermediate algebraic representation of the form $\sigma-\pi-\bowtie$ into a select-from-where clause without nested sub-queries. The only challenge in this case is to consult the database schema and get the path to reach the target table traversing the intermediate tables in the relationship tree. Here, we start from the Employee table and eventually JOIN the Leave_Details table (the target table in this case) traversing through the employee Department, Leave_Types tables to get a bag of rows corresponding to the leaves availed by the Employees.

```sql
SELECT (?,[E.employee_id], ?,[E.empl_rcd], ?,[E.eff_status], ?,[E.Designation],
      ?,[D.Department_name], ?,[LT.leave_type], ?,[LD.start_date], ?,[LD.end_date])
FROM ??[rex_employees] E
JOIN ??[rex_departments] D ON E.department_id = D.id
JOIN ??[rex_leave_details] LD ON LD.employee_id = E.id
JOIN ??[rex_leave_type] LT ON LD.leave_code = LT.id
WHERE E.id = (Select max(E1.id) from rex_employees E1) WHERE E1.id = E.id)
```

This query is a suitable initial one as the required nested clauses are not added in the WHERE clause for filtering out the actual set of the Active, latest Effective Dated rows after removing duplicates. Running this on the Rex database fetches a lot more than the actual number of rows expected from the Leave_Details table. Hence added the sub-query predicates discussed before to create the below resultant query.

```sql
SELECT (?,[E.employee_id], ?,[E.empl_rcd], ?,[E.eff_status], ?,[E.Designation],
      ?,[D.Department_name], ?,[LT.leave_type], ?,[LD.start_date], ?,[LD.end_date])
FROM ??[rex_employees] E
JOIN ??[rex_departments] D ON E.department_id = D.id
JOIN ??[rex_leave_details] LD ON LD.employee_id = E.id
JOIN ??[rex_leave_type] LT ON LD.leave_code = LT.id
WHERE E.id = (Select max(E1.id) from rex_employees E1) WHERE E1.id = E.id) AND
E1.empl_rcd = (Select max(RCD.empl_rcd) FROM rex_employees RCD WHERE
RCD.employee_id = E1.employee_id);
```

This turns out to fetch the intended outcome (Fig 7). The addition of the new nested Sub-Query: $E1.empl_rcd = (Select max(RCD.empl_rcd) FROM rex_employees RCD WHERE
RCD.employee_id = E1.employee_id);$ in the final SQL enabled the removal of old Employee records from the rows returned earlier.

---

**Figure 7: Simplified schema of an HRMS ERP database**
Figure 8: The resultant nested SQL

**EXAMPLE1:** Consider the “Employee” and “Leave Details” tables given below, where column names with suffix “_fk” indicate foreign keys.

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>emp_rcd</th>
<th>Designation</th>
<th>Eff_status</th>
<th>Departm</th>
<th>Dept_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Matt</td>
<td>1</td>
<td>Manager</td>
<td>Inactive</td>
<td>Math</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Matt</td>
<td>2</td>
<td>Professor</td>
<td>Active</td>
<td>CS</td>
<td>60</td>
</tr>
<tr>
<td>114</td>
<td>Joe</td>
<td>1</td>
<td>Trainee</td>
<td>Inactive</td>
<td>Math</td>
<td>90</td>
</tr>
<tr>
<td>114</td>
<td>Joe</td>
<td>2</td>
<td>Surgeon</td>
<td>Active</td>
<td>Medical</td>
<td>80</td>
</tr>
<tr>
<td>115</td>
<td>Rose</td>
<td>1</td>
<td>Professor</td>
<td>Active</td>
<td>Math</td>
<td>90</td>
</tr>
</tbody>
</table>

**Fig a) Employees table representation**

<table>
<thead>
<tr>
<th>Eid</th>
<th>emp_rcd</th>
<th>Leave_Type</th>
<th>From_dt</th>
<th>To_dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>Casual</td>
<td>02-Aug-2020</td>
<td>03-Aug-2020</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>Accumulated</td>
<td>26-Aug-2020</td>
<td>27-Aug-2020</td>
</tr>
<tr>
<td>114</td>
<td>1</td>
<td>Casual</td>
<td>10-May-2020</td>
<td>10-May-2020</td>
</tr>
<tr>
<td>114</td>
<td>2</td>
<td>Restricted</td>
<td>27-Aug-2020</td>
<td>28-Aug-2020</td>
</tr>
<tr>
<td>115</td>
<td>1</td>
<td>Special</td>
<td>02-Aug-2020</td>
<td>03-Aug-2020</td>
</tr>
</tbody>
</table>

**Fig b) Leave_Details table representation**

\[ \Pi(\text{Leave_Details(Eid, FK\_Eid, Employees)}) \] returns the combined details of the Employees-Leave-Data:

<table>
<thead>
<tr>
<th>Eid</th>
<th>Eff_stat</th>
<th>Em_rcd</th>
<th>Designation</th>
<th>Dept_id</th>
<th>Leave_Code</th>
<th>From_dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>I</td>
<td>1</td>
<td>Manager</td>
<td>Math</td>
<td>CL</td>
<td>02-Aug-2020</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
<td>2</td>
<td>Professor</td>
<td>CS</td>
<td>AL</td>
<td>26-Aug-2020</td>
</tr>
<tr>
<td>114</td>
<td>I</td>
<td>1</td>
<td>Trainee</td>
<td>Math</td>
<td>CL</td>
<td>10-May-2020</td>
</tr>
<tr>
<td>114</td>
<td>A</td>
<td>2</td>
<td>Surgeon</td>
<td>Medical</td>
<td>RH</td>
<td>27-Aug-2020</td>
</tr>
<tr>
<td>115</td>
<td>A</td>
<td>1</td>
<td>Professor</td>
<td>Math</td>
<td>SL</td>
<td>02-Aug-2020</td>
</tr>
</tbody>
</table>

**Fig c) Query Output with max(emp_rcd) Sub-Query**

Here, \( \Pi_{max(emp\textunderscore rcd)}(\text{Employees}) \) fetches the latest employee_record of the Employees:

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>Eff_Status</th>
<th>Em_rcd</th>
<th>Designation</th>
<th>Dept_id</th>
<th>Leave_Code</th>
<th>From_dt</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Matt</td>
<td>A</td>
<td>2</td>
<td>Professor</td>
<td>CS</td>
<td>CL</td>
<td>26-Aug-2020</td>
</tr>
<tr>
<td>114</td>
<td>Joe</td>
<td>A</td>
<td>2</td>
<td>Surgeon</td>
<td>Medical</td>
<td>RH</td>
<td>27-Aug-2020</td>
</tr>
<tr>
<td>115</td>
<td>Rose</td>
<td>A</td>
<td>1</td>
<td>Professor</td>
<td>Math</td>
<td>SL</td>
<td>02-Aug-2020</td>
</tr>
</tbody>
</table>

**Fig d) Final Output with max(emp_rcd) Sub-Query**

For the tables in Fig a & b, the final query retrieved one row each for Emp_ids 101, 114 and 115. Note that for Emp_id=101 and 114, who had 2 emp_rcds, the nested sub-query has filtered out the latest one with status=Active(A) belonging to Department_id=‘CS’
6. Results and discussions

The Experimental Database Configuration of REX is as given below.

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Database Name</th>
<th>Size</th>
<th>#Tables</th>
<th>#Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>REX</td>
<td>120 MB</td>
<td>12</td>
<td>117</td>
</tr>
</tbody>
</table>

Table 1: Experimental Database Instance

<table>
<thead>
<tr>
<th>Mode Name</th>
<th>Operating System</th>
<th>Processing Speed</th>
<th>RAM</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP15s-fr2005tu</td>
<td>Windows 10</td>
<td>2.4 GHz</td>
<td>8 GB</td>
<td>Intel i5 - 4 Cores</td>
</tr>
</tbody>
</table>

Table 2: Experimental Server Configuration

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Input DRS Count</th>
<th>Success</th>
<th>Average Time(Without FP Constructs)</th>
<th>Average Time(With FP Constructs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>4</td>
<td>4</td>
<td>475 msecs</td>
<td>315 msecs</td>
</tr>
</tbody>
</table>

Table 3: Results Summary

The SQLs generated by the REX framework were executed against the REX database instance a) first without the Functional Programming constructs b) then with the Functional Programming constructs (using Monad comprehension and Foldr) discussed in this work. By incorporating the Functional Programming constructs the speed of execution improved by 33%.

7. Future work

Further research can be performed to establish the Turing machine compliance of the intermediate Language described in this work. We can go a step further by applying Lambda Context calculus to ensure the effectiveness of translation. Techniques for SQL optimization by applying techniques at intermediate level can be pursued. The correspondence between finite form of structural recursion and relational algebra makes it possible to apply optimization techniques directly into the language. Further research can be conducted to verify if application of qualifier exchange rule provides the means to reorder filters and joins so that query rewrite is managed with the help of indices.

8. Conclusions

This work extended the earlier approaches for automatic query creation by adopting advanced concepts from the functional programming domain. An intermediate language centred on structural recursion is constructed for performing the required transformation operations and for representing data structures used in processing the data spanning across different levels of a tree. Adoption of structural recursion is the most significant design choice in the intermediate representation as it supports type-based design, represents algebraic and extended data types, provided an initial skeleton on which transformations can be performed and supports nesting. This work also described how on working with nested intermediate structures, Monad comprehensions provided the necessary syntactic sugar and was helpful in combining a wide range of translation constructs, such as transformation rules and state management, exception handling or managing input-output to eventually return the desired SQL query. The recursion Combinator foldr effectively combined monad comprehensions and Combinators, providing the necessary platform for merging fixed and varying components of the intermediate
representation. The peculiarity of the notion of monads is that it comes with just enough internal structure to represent the query calculus. The resulting monad comprehension calculus eventually leads to a form of query representation that corresponds to the core structure inherent in a query. The single uniform formal framework designed for translating NL to SQL effectively combined all stages of the query synthesis process and produced deeply nested queries. The translation framework eventually had a program that combined all these techniques which eventually transformed requirement statements into nested SQLs.

9. Bibliography

1 Bags: Collection of data where repetition of elements is allowed (unlike sets).

2 Combinator: A Combinator is a λ-calculus expression to represent primitive functions which has no free variables. A Combinator represents closed expressions (no free variables) of a language and corresponds to axioms of a deductive system.

3 Orthogonal design: (or Othogonality) in programming language design is the ability to use various language features in arbitrary combinations in such a way that independent concepts are kept independent and not mixed together to avoid complexity. It ensures that modifying the technical effect produced by a component of a system does not create or cascade side effects to other components of the system.

4 Initial Algebra: Algebra of abstract data types and their constructors plus the rules and functions associated with these data types.

5 Lambda Calculus: (also denoted as λ-calculus) is a formal system in mathematical logic to express computation based on function abstraction and application by variable binding and substitution. It is a universal model of computation and can simulate any Turing machine.

6 Fixed-point Combinator: Fixed-point Combinators are used for implementing loops in Lambda calculus. They are also used to implement recursion without calling the function name recursively but by applying the function to itself with a new set of values for its bound variables every time when it is (re)applied.

7 Structural Recursion: Programming paradigm that enables to perform recursion on objects made from user defined data types. Recursion on dynamic data structures such as Lists and Trees where data to be treated are defined in recursive terms. Structural recursion over lists has been known under the function names fold or reduce.

8 Comprehensions: Comprehension offer a concise way of creating a data structure from one or more iterators. Comprehensions make it easier to combine loops and conditional statements with less verbose syntax.

9 Monad: Monads provide a framework to combine a wide variety of programming paradigms, such as managing state, exceptions, or input-output. It has a return operator that creates values, and a bind operator to link the actions in the pipeline; and its definition follows a set of axioms called monad laws, all these are mandatory for the composition of actions in the pipeline to work properly. The final result is the outcome of the entire unit.

10 Monad Comprehensions: In computer science Monad and Monad Comprehension are inter-changeably used.

11 Fold: In functional programming, fold refers to the use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.

12 Foldr: Foldr stands for fold-right while operating.

13 Generator: A generator is a sequence creation object and often the source of data for iterators and allows iteration through potentially huge sequences without creating and storing the entire sequence in memory at once. It is different from a normal function which has no memory of previous calls and always starts at its first line with the same state. But generators keeps track of where it was the last time it was called and returns the next value.

References


A novel framework for synthesizing nested queries in SQL from business requirements language


