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

Mathew George[1], Dr. Rohini V[2]

[1,2]Department 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 resultingin'nested'queries havealways been a challenge and was not effectively handled. This work proposes a framework for translating requirement statementsresulting inthe construction of nested Queries. While translating nested scenarios; often thereis a need to create sub-queriesthat execute inpipeline orin 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 transformationson theintermediate expressions involving these,more advancedprogramming constructs are needed.This work recommends the use of advanced language constructs and adoptsfunctional 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 formof Rewrite systems that provided a platform for intermediate representationand offered astandard method for modelingcomputation [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 algebras[4] of a core functional language is followed and intermediate representationsthatsuits the demands of nested query generation is subsequently developed.Advanced type systems areneeded in the design of an intermediate language for representing nested queries.Applying relational query processing rules alone will notbe sufficientto represent and to perform transformations on these extended type systems.

For generating nested queries, it is imperative to define acalculus and a language that can represent comprehension syntax and perform the required operations as relationalcalculus does to relational query languages. Its main processing requirement is to perform structural recursion[7] on bulk data types like bags[1] 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, comprehensions[8] and basic Combinators[2]togetherin effectcan represent and meet the transformation requirements of the intermediate language.The comprehension calculus providesthe means to canonically represent and effectively reason about complex predicates, including quantifiers, and collection processing.Advanced programming constructs like Monad Comprehensions[10] and Folds[11] 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 programmingtechniques can provide the adequate framework for the automatic derivation of SQLs from Data Requirement Specifications

The organization of the paper is asfollows:A Novel framework toThe next section discusses the 'RELATED WORK'and the progress made by research communityin synthesizing nested queries.The key technical contributionsand their inclusionin the transformation frameworkaregiven in theOVERVIEWSection. Detailed concepts and their relevancefollowed bya theoretical walk throughcan be seen in the section onCURRENT WORK. A motivating exampleand the steps for translating a sample nested scenario,is given in the section named 'A CASE STUDY'.The SCOPE

FOR FUTURE WORKand the CONCLUSIONS are described in the last two sections. A BIBLIOGRAPHYof terms andthe details of literature referred can be found inthe REFERENCES section.

2. Relatedwork

Yaghmazadeh N. and Dillig I. (2017)proposed a typeand database content driven synthesize-repair framework [1] for synthesizingSQLs from Natural language statements. Rewrite methods and Inference Rules based transformationsarecentral to their work.The method suggested for synthesizing nested queries isto repeat the same process used for generating themain query. The pipelining and dynamic re-organization required whilecombining sub-queries cannot be brought outeasily 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 algebrasasintermediaterepresentation which can be transformedby 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 setswhile SQL is primarily based on bags [2] and the query algebra operators are in fact abstract representation of underlying proceduresimplemented by the query engine.Query predicates were viewed as annotations to algebraic operators and were not part of the translation or compilation phasebutwere 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 foran 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 andsubsequently 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 forSQL 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 translationscenario, 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 communitiesat the intermediate language levelto 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 followsthe 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 programmingis 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 contextsand 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. 268

Key Concepts

The key transformation techniquesdiscussed 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 whiletranslatingfrom theAbstract Query Language into an equivalent SQL query.
3. Application of higher-order Combinators likefolds (foldr[12]) for abstract representation and as a means to augmentrecursiveprocessing initiated through Fixed-point Combinators[6].

System Architecture:This work takes over once an initial query sketchis 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 [12] is extendedfurther by implementing the advanced techniques discussed here.

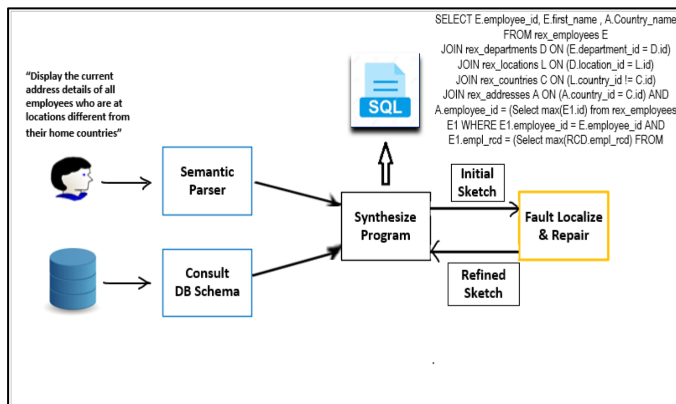

Figure 1: System Architecture

4. Current work

Road map:

The componentsof the core intermediate languageis defined first, followed by the transformations need to be performed on the expressions created using this language for achieving the translation.The entire workis centered onthe application of functional programming techniques and scaling itfurther by using higher-order functions and their associated operations as when needed. The pipelining techniques for effectively combining nested expressionsare described next. The benefit of usingMonad Comprehensions and foldrsand theireffectiveness in chaining and pipelining different translation components are keysto the implementation of the concepts proposed and applied subsequently.Finally, a case study is presented to evaluatethe 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 recursionof 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 waysbut 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 (thatcaters 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 bagsbyplacingthem 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 techniquesfound around first class functions.

Lambda Calculus combined with Comprehensions

Functional languages are usually based on lambda calculus[5]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 Combinatorswould 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 fashionedto take their own specific inputs and passes intermediate results.

The Combinator sub-language, can be extend by applying the syntactic sugar - comprehensions which providesbetter 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[3] combined and freelyrearranged 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 algebrasexposes its own limitation: especially when temporary results are communicated between operators andsince these are designed separatelyto consume their own inputs, producestheir intermediate results bringing out the necessityfor 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[10] provide a framework for bundling / structuring the semantic representation of features such as state, exceptions and continuations[8]**.** Monad Comprehensionsare recommended for use at the intermediate language level to bundlerelated components and features, byarraying themin 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 Comprehensionsprovide 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 relateddata modelcommunities can be applied on thisintermediate 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 valuesto construct a return valueeventually.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:
fold f z xs

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

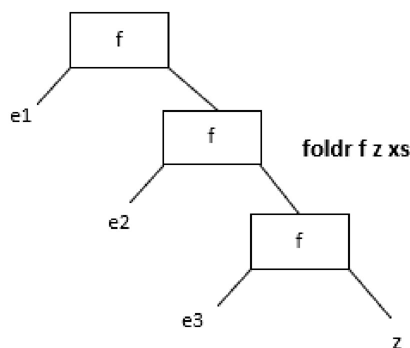

Figure2: Representation of foldr

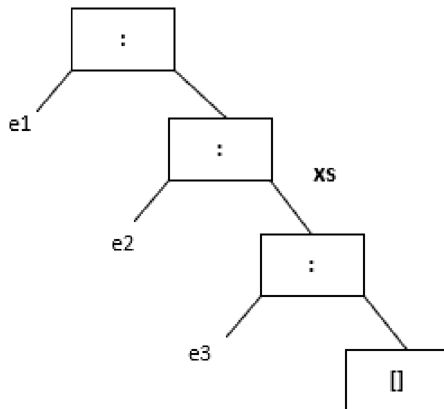

Figure3: Foldrimplementation

Here$e_{1,}$ e_2, $e_3$represents sub-query expressions and [] represents a list object

The head expression *e*(*selet-from* part) of an SQL statement is defined as:
e→| 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
$[]^{\tau}$ | $(:)^{\tau}$ = Constructors
σ = selection
τ = algebraic data type over a relationranged by the below expression. Let *l* range over a set of labels
τ ::=list | set | bag | unit | int | boo1 | string | real | τ→τ

The domain of a type τitselfcan be deemed as an algebra.

Figure 4:Intermediate Language Definition

The *select-from-where* block can be represented in the intermediate query language. The intermediate mapping constructQ is represented as:

Q *(select e from* e_1 *as* $x_1, \ldots,$ e_n *as* x_n *where p)* $= [Q\ e \mid x1 \leftarrow Q\ e_1 \ldots\ e_n \text{ as } x_n \leftarrow Q\ e_n,\ Q\ p]^{bag}$(the x_i'sappear free in *e* and *p*).

Aquery 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 $[e \mid q_1,\ldots,q_n]^{\tau}$ the predicates q_i are either generators[13]$v \leftarrow q$or filters (expressions resulting in type bool). A generator $q_i = v \leftarrow q$ sequentially bindsvariable v to elements of its range q; v is bound in $q_{i+1}, \ldots, q_n$ and e. The bindingof v is propagated until a filter evaluates to False under the binding. The result of evaluating e is collected in the list construction $(:)^{\tau}$.

$\text{map}^{T} f\ s = [f\ x \mid x \leftarrow s]^{T}$
$\text{filter}^{T} p\ s = [x \mid x \leftarrow s,\ p\ x]^{T}$
$\text{cross}^{T\sigma} s\ t = [(x, y) \mid x \leftarrow s,\ y \leftarrow t]^{T}$
$\text{join}^{T\sigma} p\ f\ st = [fx\ y) \mid x \leftarrow s,\ y \leftarrow t,\ p\ x\ y]^{T}$
$\text{semi-join}^{T\sigma} p\ s\ t = [x \mid x \leftarrow s,\ [p\ x\ y] \mid y \leftarrow t]^{exists}$
$\text{anti-join}^{T\sigma} p\ s\ t = [x \mid x \leftarrow s,\ [\neg p\ x\ y \mid y \leftarrow t]^{all}]^{T}$
$\text{nest-join}^{T\sigma} p\ f\ s\ t = [\ [fx\ y \mid y \leftarrow t,\ p\ x\ y]^{\sigma} \mid x \leftarrow s]^{T}$
$\text{max}^{T} f\ s = [\ fx \mid x \leftarrow s]^{max}$ (max ∈{ min, exists, all, sum})

Figure 5: Algebraic Combinators (Monad Comprehensionbased definitions)

5. A case study

Prepare the SQL for the following DRS statement: "Retrieve***leave*** details of all **employees**basedon their***latestemployment records***".

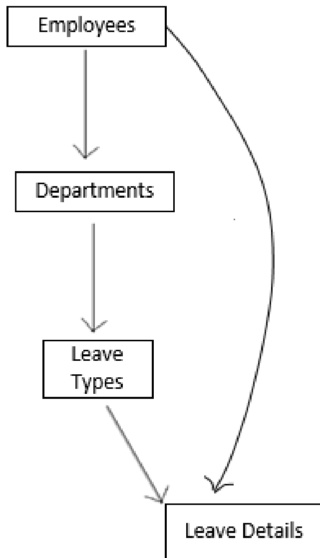

Fig6: DB Table Relationship

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 acomplex 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 $[x \mid x \leftarrow xs,\ p\ x]^{\tau}$is similar to the relational selection $\sigma_p.xs$ but more generalised to represent any data type of τ and can be shown as:

$[x \mid x \leftarrow xs,\ p\ x]^{\tau} \simeq \sigma_p.xs$

The nested comprehension is represented as:

$[f\ x \mid x \leftarrow xs,\ [gx = h\ y \mid y \leftarrow ys,\ py]^{exists}]^{set}$

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 ewhere e.id in (select e.id
 from Leave_Details
where a.employee_id = e.id);
```

The *select-from-where* combination is identical to a comprehension: the `from` clause corresponds to a sequence of generators[12], and the predicates in the where clause corresponds to filters. Finally, the *select* clause represents the comprehension's head expression. Use of the *distinct* modifier would transform a *bag* into a *set* as the result monad[9] [2].Moreover, nested SQLs operate in a streaming (or pipelined) mode.SQL execution benefits from streaming since objects are addressed and loaded from the persistent storage only once.

Functional composition will be the most preferred way forbuildingnested queries.Structural recursion (and an implementation of it in the form of *foldr*) provides the principal way for implementing functional abstractionover values of an initial algebraic data type τ. The Combinators may be re-expressed by ***foldr*** directly [2]. The foldr-based program schememay then be used as a template to derive an actual typically imperative storage access program due to the simple linear recursion scheme represented by ***foldr*** [2].

$\text{nestjoin}^{\tau\sigma} p\ f\ s\ t = \text{foldr}^{\tau}(\lambda x xs.(\text{foldr}^{\sigma}(\lambda y ys.\ \text{if exists}^{\sigma}\ (p\ y)\ t\ \text{then}\ y :^{\tau} ys\ \text{else}\ ys)\ []^{\sigma}) :^{\tau} xs\ t)\ []^{\tau} s$

As translation schemecan translate Monad Comprehensions into nested ***foldr****expressions*,which on executingthequeries constructed from these expressions, the query engine will tagfor 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.

$nestjoin^{\tau\sigma} p\ f\ s\ t = [foldr^{\tau}(\lambda e_id\ employees.(foldr^{\sigma}(\lambda ld_id leave_details\ s.\ if\ exists^{\sigma} max(e1_id)\ then\ max(e1_id)) :^{\tau} employees\ t\ )]^{bag} \boldsymbol{emp_rcd_subquery(s)}[\,]^{\tau}$

The **emp_rcd_subquery** predicate:

$s = \lambda x\ y{:}p\ y\ \&\&q\ x\ y$

$\rightarrow$

$p = [\,]^{\tau} a.employee_id = (foldr^{\sigma}(\lambda e1.id)\ employees1(\lambda e1_id\ employees1\ t.\ if\ exists^{\sigma} e1.id{=}e.id\ then\ max(e1.id){:}^{\tau})$

&&

$q = [\,]^{\tau} e2.empl_rcd = (\lambda e2_id\ employees2\ s.if\ exists^{\sigma} e2.id = e1.id and\ e2.eff_status{=}\text{'Active'}\ then\ max(e2.empl_rcd){:}^{\tau})[\,]^{\tau}$

byappending nested emp_rcd_subquery predicate q.

Fig7: 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 Combinatorfoldr, 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, leave details. We can express this query as:

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

It is a relatively easy task to map the intermediate algebraic representation of the form σ-π-⋈ into a select-from-whereclause 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_Detailstable (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.

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 suitableinitial one as the required nested clauses are not added in the WHERE clause for filtering out the actual set of theActive, latest Effective Dated rowsafter removing duplicates. Running this on the Rex database fetches alot 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.

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 fetch the intended outcome (**Figd**).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 form the rows returned earlier.

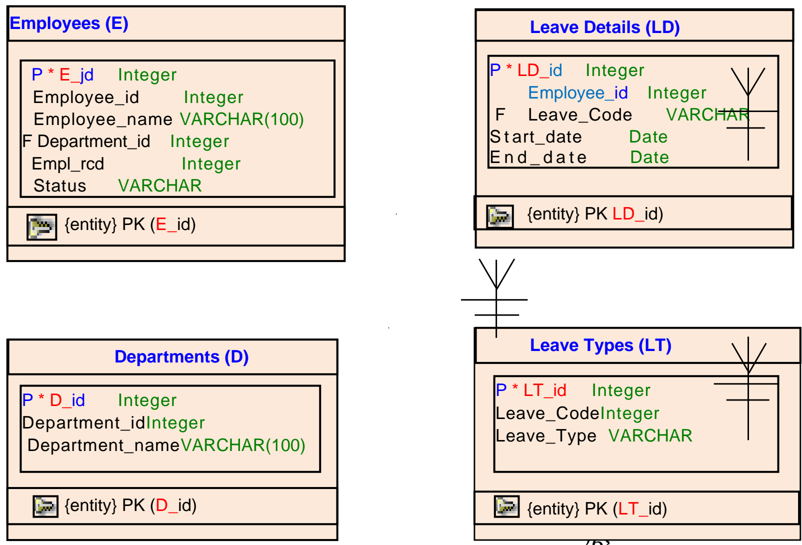

Figure 7: Simplified schema of an HRMS ERP database

```sql
SELECT E.employee_id, E.first_name, 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));
```

Figure 8: The resultant nested SQL

EXAMPLE1: Consider the “Employee” and “Leave_ Details”tables given below,where column names with suffix “_fk" indicateforeign keys.

Eid	Name	em p_rcd	Designatio n	Eff_statu s	Departm ent	Dept_id
101	Matt	1	Manager	Inactive	Math	90
101	Matt	2	Professor	Active	CS	60
114	Joe	1	Trainee	Inactive	Math	90
114	Joe	2	Surgeon	Active	Medical	80
115	Rose	1	Professor	Active	Math	90

Fig a) Employees table representation

Eid _*fk*	emp_ rcd	Leave_Type	From_dt	To_dt
101	1	Casual	02-Aug-2020	03-Aug-2020
101	2	Accumulated	26-Aug-2020	27-Aug-2020
114	1	Casual	10-May-2020	10-May-2020
114	2	Restricted	27-Aug-2020	28-Aug-2020
115	1	Special	02-Aug-2020	03-Aug-2020

Fig b)Leave_Details table representation

Π(Leave_Details$_{Eid\text{-}fkEid}$*Employees)* returns the combineddetails of the Employees-Leaves-Data:

Eid _*fk*	Eff_s tatus	Em p_rcd	Designation	Depa rtment	Leav e Code	From_dt
101	I	1	Manager	Math	CL	02-Aug-2020
101	A	2	Professor	CS	AL	26-Aug-2020
114	I	1	Trainee	Math	CL	10-May-2020
114	A	2	Surgeon	Medi cal	RH	27-Aug-2020
115	A	1	Professor	Math	SL	02-Aug-2020

Fig c)Query Output with max(emp_id)Sub-Query

Here, *Π*$_g$*max(emp_rcd)(Employees)* fetches the latest employee_record of the Employees:

Eid	Nam e	Eff _Status	Em p_rcd	Designatio n	Dept _id	Leav e Code	From_dt
101	Matt	A	2	Professor	CS	CL	26-Aug-2020
114	Joe	A	2	Surgeon	Medi cal	RH	27-Aug-2020
115	Rose	A	1	Professor	Math	SL	02-Aug-2020

Fig d)Final Outputwith max(emp_rcd)Sub-Query

For the tables in**Fig a & b**, the final query retrievedone row eachforEmp_ids 101, 114 and 115.Note that for Emp_id=101 and 114, who had2emp_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.

Database Type	Database Name	Size	#Tables	#Columns
Postgres	REX	120 MB	12	117

Table 1: Experimental Database Instance

Model Name	Operating System	Processing Speed	RAM	Processor
HP15 s-fr2005tu	Windows 10	2.4 GHz	8 GB	Intel i5 - 4 Cores

Table 2: Experimental Server Configuration

Query Type	Input DRS Count	Success	Average Time(Without FP Constructs)	Average Time(With FP Constructs)
Nested	4	4	475 msecs	315 msecs

Table 3: Results Summary

The SQLs generated by the REX framework were executed against the REX database instance a) first without the Functional Programming constructsb) 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 constructedfor 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, providedaninitial skeleton on which transformations can be performed and supports nesting.This work also described how onworking 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 programthat combined all these techniques which eventually transformed requirement statementsinto 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 denotedas 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 ondynamic 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:** Comprehensionsoffer a concise wayof 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 aframework 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 monadlaws, 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 potentiallyhuge sequences without creating and storing the entire sequence in memoryat once. It is different from a normal function whichhas 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 timeit was called and returns the next value.

References

1. N.Yaghmazadeh and Y.Wangand I. Dillig, and T. Dillig, “Type and Content-Driven Synthesis of SQL Queries from Natural Language,” Computer Science - Databases, Computer Science –Programming Languages, eid = arXiv:1702.01168, Feb 2017. Available: https://arxiv.org/abs/1702.01168.
2. Muxamediyeva, D. K. "Properties of self similar solutions of reaction-diffusion systems of quasilinear equations." International Journal of Mechanical and production engineering research and development

(IJMPERD) 8.N (2018).
3. T. Grust and M. H. Scholl, “How to comprehend queries functionally,” J. Intell. Inf. Syst., vol. 12, no. 2, pp. 191–218, 1999.
4. V. Breazu-Tannen, P. Buneman, and S. Naqvi, “Structural recursion as a query language,”DBPL3 Proc. third Int. Work. Database Program. Lang. bulk types persistent data, no. August, pp. 9–19, 1992.
5. M. Bognar, “Contexts in Lambda Calculus,” Vrije Univ.Amsterdam, Thesis, pp. 1–236, 2002.
6. P. Buneman, M. Fernandez, and D. Suciu, “UnQL: A
7. Querylanguage and algebra for semi-structured data based onstructural recursion,” VLDB J., vol. 9, no. 1, pp. 76–10, 2000.
8. DEVI, MEENU, S. R. Verma, and M. P. Singh. "An efficient method of bounded solution of a system of differential equations using linear legendre multi-wavelets." Int. J. Math. Comp. App. Res 4 (2014): 2249-8060.
9. B. Carpenter. Type-logical semantics. MIT press, 1997.
10. R.S.Bird (1987). An Introduction to the Theory of Lists In M. Broy(Ed.), Logic of Programming and Calculi of Design. NATO ASI Series. Springer Verlag, vol. 36,p. 5–42.
11. KRYUCHKOV, VASSILIY, et al. "Investigation of dynamic motion processes of modernized uav using mathematical model of numerical simulation." International Journal of Mechanical and Production Engineering Research and Development 10.2 (2020): 535-554.
12. P. Wadler, “Comprehending Monads,” Math. Struct. Comput. Sci., vol. 2, no. June 1990, pp. 1–38, 1970.
13. Gill, J. Launchbury, and S. L. P. Jones, “Short cut to deforestation,” no. Section 4, pp. 223–232, 1993.
14. Islam, MdAsraful, and Payer Ahmed. "Prediction of the Population of Bangladesh Using Logistic Model." International Journal of Applied Mathematics & Statistical Sciences (IJAMSS) 6.6 (2017): 37-50.
15. P. van Leeuwen, “λ-Calculus Syntax’s Definition and Completeness As Graph Database Querying Language.pdf.” Semantic Scholar, pp. 1–8, 2018.
16. Jain, A. B. H. I. N. A. V., and M. O. N. I. K. A. Mittal. "Haar wavelet based computationally efficient optimization of linear time varying systems." International Journal of Electrical and Electronics Engineering (IJEEE) 3.3 (2014): 11-20.
17. P. Buneman, M. Fernandez, and D. Suciu, “UnQL: A query language and algebra for semistructured data based on structural recursion,” VLDB J., vol. 9, no. 1, pp. 76–110, 2000.
18. HosuI., IacobR., BradF., Ruseti, S. and RebedeaT. (2018). “Natural Language Interface for Databases Using a Dual-Encoder Model”. Proc. 27th Int. Conf.Comput. Linguist., pp. 514–524.
19. P.Selinger(2013), “Lecture Notes on the Lambda Calculus”,Department of Mathematics and Statistics,Dalhousie University, Halifax, Canada.