An SQL Frontend on top of OCaml for Data Analysis

Yan Dong  
National University of Singapore  
Singapore, Singapore  
dong.yan@u.nus.edu

Yahui Song  
National University of Singapore  
Singapore, Singapore  
yahuis@comp.nus.edu.sg

Wei-Ngan Chin  
National University of Singapore  
Singapore, Singapore  
chinwn@comp.nus.edu.sg

ABSTRACT

SQL is a widely adopted programming language for relational databases. Its ability to concisely process data sets makes it suitable as a complementary construct for data analysis in general-purpose programming languages. This work presents SelectML, which integrates the Select Query capabilities directly into the Ocaml language, where a query is a first-class language construct. Query expressions are in declarative syntax, where programmers can perform filtering, ordering, and grouping operations on data sets with a minimum of code. SelectML retrofits the queries into pure OCaml, based on the formally defined translation schema. Our implementation allows SelectML to be a flexible framework for customizing the types of inputs, outputs and operations of the queries. SelectML deploys query plan optimizations to make query executions more efficient. We also discuss the potential of connecting to databases from SelectML.

CCS CONCEPTS

• Software and its engineering → Domain specific languages;  
  Software prototyping; • Computing methodologies → Model development and analysis.

KEYWORDS

SQL, Select Query, Databases, OCaml, Functional Programming, Formal Semantics

ACM Reference Format:

1 INTRODUCTION

SQL (Structured Query Language) is a declarative programming language widely adopted in modern relational databases for managing, storing, and manipulating data. The Select Query is the most common operation in SQL, used to retrieve data from the databases and perform computations on the data. The common Select Query consists of clauses including SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. With Select Queries, users can specify expected results without giving detailed computation steps. However, for databases to execute a Select Query, a precise sequence of instructions has to be given instead of a declarative specification. The query plan (or execution plan) is an imperative language that is automatically generated by databases to describe the concrete steps to achieve the query specification. Databases are also responsible for finding the optimal query plan for a query by some given criteria [8] [15] such as soundness and performance.

Most SQL databases support a query plan level optimization for the Select Query, which may reduce the original query’s execution time, space consumption, or complexity. Such techniques can bring benefits to general-purpose programming languages, which are equipped with Select Query constructs, to enable a state-of-art level query plan optimization. For example, some functional languages like Haskell and Scala support a syntactic sugar called list comprehension for specifying the expected list of data in a declarative way similar to the Select Query.

On the other hand, since the Select Query conforms well with the functional programming paradigm by working on immutable data, it seems to be an excellent complementary to data analysis for languages, such as OCaml, that lack a similar language construct. Therefore, in the paper, we present SelectML, as a language frontend on top of OCaml (or OCaml + Select Expression per se), with support of the Select Expression as a new language construct for efficient data analysis.

1.1 Declarative Data Processing

Imagine a scenario, we have a list of food orders for the past few years, and we want to sort the monthly incomes in a decreasing order for the year 2021. In OCaml, we might write the query as:

```
1  type order = ( id: int; price: float; month: string )
2  let group f l =
3      let eq a b = compare (f a) (f b) = 0 in
4      List.to_seq l |> Seq.group eq
5      |> Seq.map List.of_seq |> List.of_seq
6
7  let income_by_month : order list -> (string * float) list =
8      fun orders -> orders
9          |> List.filter (fun o ->
10            o.month >= "2021-01" && o.month <= "2021-12")
11          |> List.sort (fun a b -> compare a.month b.month)
12          |> group (fun o -> o.month)
13          |> List.map (fun l ->
14            List.fold_left
15              (fun (m, income) o ->
16                (m, o.price) <+ income))
17              (List.hd l).month, 0.0) 1
18          |> List.sort (fun (_, s1) (_, s2) -> compare s2 s1)
```

Figure 1: Data Transformation in a Functional Manner

Although the process of data transformation is clearly expressed in a functional manner, shown in Figure 1, it can be simplified with the support of Select Queries. We can rewrite the code in a more declarative and concise way by using the Select Expression, shown
in Figure 2. It is not hard to observe that it’s much easier to specify the expected result using SQL, especially for programmers who are familiar with SQL databases or working in data analysis.

```
let income_by_month : order list -> (string * float) list =
fun orders ->
  SELECT o.month, {sum o.price} FROM o <- orders
WHERE o.month >= "2021-01" && o.month <= "2021-12"
GROUP BY o.month ORDER BY (sum o.price) DESC
```

**Figure 2: Data Transformation using Select Expression**

In this work, we focus on in-memory computation, where all the calculations are entirely in computer memory (e.g., in RAM). We discharge the possible interactions with the databases using the standard Ocaml libraries, such as 'ocaml-caqti' [4].

1.2 System Overview

*SelectML* is a SQL language extension of OCaml, which is implemented by modifying the OCaml compiler front-end (version 4.14), particularly by extending the parser to recognize the Select Expression, supporting the typing/translation of the Select Expression to the compiler intermediate representation.

![Figure 3: An Overview of OCaml Frontend](image)

Figure 3 gives an overview of the compilation phases in the OCaml front-end. The front end takes the OCaml source program as the input. Firstly, in the parsing phase, the source program is parsed into the Parsetree (i.e., the abstract syntax tree). Secondly, in the typing phase, the Parsetree is annotated with type expressions and becomes Typedtree (the typed version abstract syntax tree). Lastly, the Typedtree is translated to the Lambda intermediate representation and passed to the OCaml back-end for generating bytecode and native code.

![Figure 4: An Overview of SelectML](image)

As shown in Figure 4, *SelectML* works by adding the Select Expression as a new language construct to the Parsetree. For other OCaml constructs, they are typed to Typedtree just as usual, while the phase for the Select Expression is different. The query plan, an imperative language that describes execution steps of a declarative query, is generated after type-checking of Select Expression. Then the query plan is optimized and translated to the plan-free Type-tree. The constructs and steps marked as darkred are our main contributions.

2 LANGUAGE DESIGN

This section presents the design decisions of *SelectML*. SQL statements are categorized into four groups: DQL (Data Query Language), DDL (Data Definition Language), DML (Data Manipulation Language), and DCL (Data Control Language) [23]. DQL, represented by `SELECT`, is used to specify the expected results from tables in the database. DDL, represented by `CREATE, ALTER, DROP`, is used to define the schema of databases, tables, indexes, etc. DML, represented by `INSERT, UPDATE, DELETE`, is used for adding, modifying, and deleting data in databases. DCL, represented by `GRANT, REVOKE`, is used to control the access to the database.

It is easy to find correspondence for DDL and DML in general-purpose languages. For example, variable, module declarations are analogous to `CREATE` command in DDL, assignments modifying the data are analogous to `INSERT, UPDATE` commands. DCL does not have correspondence, as access control is not involved in most programming languages. `SELECT` queries, as a representative for DQL, will be retrofitted to pure OCaml, as the main focus of this work.

2.1 Features

The retrofitted Select Query in *SelectML* is called the Select Expression. It can be used exactly the same way as other OCaml expressions, i.e. as function arguments, return values, or operands to input operators, shown in Figure 5.

```
1 (\# as arguments \#)
2 f 123 (SELECT x FROM x <- xs)
3 (\# as return values \#)
4 let g xs = SELECT x FROM x <- xs
5 (\# as operands \#)
6 x :: SELECT y FROM y <- ys
```

**Figure 5: Usages of the Select Expression**

The Select Expression provides a declarative manner of specifying the expected result. It covers common operations for data processing, including mapping, filtering, grouping, and sorting. As programmers don’t need to worry about the concrete steps of the data transformation, it gives the compiler spaces for query plan optimizations. Currently, SelectML provides some core SQL features for data processing [13], including FROM clause for inner joins, WHERE and HAVING clauses for filtering, GROUP BY clause for grouping and aggregation, and ORDER BY clause for sorting, cf. Section 2.2.

2.1.1 Features Not Covered. SQL provides various of syntax for the Select Query, which corresponds a wide range of query scenarios. In SelectML, we focus on a core set of syntax. Those query scenarios that need to be expressed using specialised SQL syntax can be implemented using normal OCaml expressions as a fallback option. This first scenario is the ability of query plan optimisations is therefore reduced since there is an information loss in expressing those specialised query syntax using a generalised syntax.

```
/* SQL */
1 SELECT ... WHERE EXISTS (SELECT ...)
2 SELECT ... WHERE col IN (SELECT ...)
```

**Figure 6: Sub-queries**

Figure 6 gives examples of the information loss of using usual OCaml expressions. Mainstream databases can optimize the sub-queries into semi-joins at the plan level knowing the meaning...
of syntax EXISTS and IN. However, currently, SelectML can only express the same logic fallback to OCaml. Thus the compiler does not know the semantics of List.length and List.mem, thereby cannot perform optimization on these sub-queries. Figure 7 gives another example of the information loss. The logic for common table expressions (also known as WITH clause) is expressed with let in syntax in OCaml. Other features absent in SelectML, like outer joins, window functions, will be discussed in Section A.

1 /* SQL */
2 WITH t AS (SELECT ...)
3 SELECT ...

Figure 7: Common Table Expressions

2.2 An Informal Syntax

The syntax for SelectML is presented in Figure 8. SelectML expressions consist of regular OCaml expressions including constants, variables x, tuples (e₁,...,eₙ), list, and functions. On top of OCaml expressions, there are two new syntactic constructs: 1) The Select Expression q; 2) The aggregation (e₁,e₂) where e₁ is the aggregate function, and e₂ is the argument.

Expressions e ::= ... | q | (e e)  
Select Expressions q ::= SELECT [DISTINCT] e [FROM s] | WHERE e | GROUP BY e | HAVING e | ORDER BY o  
Source Expressions s ::= x → e | (x₁,...,xn) → e | s,s  
Order Expressions o ::= e | ASC | DESC | USING e | o,o

Figure 8: An Informal Syntax of SelectML

For a Select Expression q, the SELECT clause is mandatory, together with optional DISTINCT keyword, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses. Details for each clause are elaborated in the following sections. The capitalised keywords are chosen for SelectML to avoid compatibility issues with existing codebases, as variable names like FROM, desc have been frequently used in the OCaml standard libraries and the compiler itself.

2.2.1 The FROM Clause. Input lists are specified in source expressions s in the FROM clause. The syntax for specifying the table alias is changed in SelectML by turning source AS x to x ← source, which makes it easier to resolve parser conflicts. x in source expressions is a shorthand for x ← x. The type of source is expected to be 'a list. A direct comparison of the SQL syntax and SelectML is given in Figure 9.

1 /* SQL */
2 SELECT x, y FROM xs AS x, ys AS y;
3 (* SelectML *)
4 SELECT x, y FROM x ← xs, y ← ys;

Figure 9: Examples of the FROM Clause

2.2.2 The WHERE and HAVING Clauses. The WHERE and HAVING clauses both function as a filter on the input data. The WHERE clause filters data coming from the FROM clause, while the HAVING clause is designed to filter aggregated data from the GROUP BY clause.

1 SELECT x FROM x ← xs WHERE f x;
2 SELECT (h x) FROM x ← xs HAVING g (h x);

Figure 10: WHERE and HAVING Clauses in SelectML

Expressions behave as predicates in the WHERE and HAVING clauses, which means f x, g (h x) in the examples (Figure 10) are expected to have type bool. Notice OCaml does not allow NULL value, the predicate can only be evaluated to true or false. But in SQL, boolean values can be NULL. This is 2-value logic versus 3-value logic, which are proved equivalent in terms of expressiveness [16].

2.2.3 The GROUP BY Clause and Aggregation. Expressions listed in the GROUP BY clause serves as the group keys. The GROUP BY clause is used to group those rows with the same group keys, and aggregation is performed on each group using certain aggregate functions (Figure 11). When there is aggregation in SELECT clause, HAVING clause, or ORDER BY clause but with no GROUP BY clause, the whole input will be viewed as one group.

1 /* SQL */
2 SELECT x, COUNT(y) FROM t GROUP BY x;
3 (* SelectML *)
4 SELECT x, (count y) FROM x ← t GROUP BY x;

Figure 11: Examples of the GROUP BY Clause and Aggregation

In SQL, several common aggregate functions are provided by default, i.e. AVG, COUNT, SUM, MIN, MAX. To help recognise aggregation in SelectML, the syntax (e₁,e₂) is used to denote the application of aggregate functions, where e₁ is the aggregate function, and e₂ is the argument column. Some databases (e.g., PostgreSQL) support user-defined aggregate functions, where users may supply the initial argument column. Some databases (e.g., PostgreSQL) support user-defined aggregate functions, where users may supply the initial argument column.
count, ...) have the type (‘a, ‘b) agg, which is now a built-in type of the compiler. For an aggregation (f x), f is expected to have type (‘a, ‘b) agg; x is expected to be in type ‘a; the result type should be ‘b.

Function mkagg is for creating user-defined aggregate functions (Figure 13), users have to supply the initial state (init1), the transition function (update), and the result function (final1). Usage of the syntax \{e1, e2\} is only allowed within the context of Select Expression (line 6 to 7), otherwise, an error will be reported (line 13 to 15). Aggregate functions can be used manually outside the Select Expression, cf. line 9 to 11 in Figure 13.

2.2.4 The ORDER BY Clause. The ORDER BY clause is used to sort the data after the HAVING clause is handled. The order expressions o in the ORDER BY clause serve as the sorting key. An order expression can be given an optional order direction, which is either ASC for ascending order, DESC for descending order, or USING for using e for as the compare function. The compare function is expected to have type ‘a -> ‘a -> int which is the same as the stdlib function compare. Ascending order is assumed by default if no direction is specified. In Figure 14, the output data will be sorted by x in the ascending order, then by y in the odd-number-first order. The result is shown on line 10.

```ocaml
let odd_first a b =     
  let x = a mod 2 = 0 in     
  compare x y;;
SELECT x, y from \{("a", 2); ("a", 3); ("b", 4); ("b", 5)\}     
ORDER BY x ASC, y USING odd_first;;
```

Figure 14: Examples of the ORDER BY Clause

2.2.5 The SELECT Clause and Typing. Expressions listed in the SELECT clause become the output data. If keyword DISTINCT is present, then duplicate rows will be removed from the output. In general cases, if the expression in the SELECT clause has type ‘a, then the type of the Select Expression will be ‘a list, cf. Figure 15 from line 1 to 8. Nevertheless, when it can be determined that there is exactly one row in the output, cf. Figure 15 from line 10 to 17, it is unnecessary to put the result into a list, hence the result type will be ‘a instead of ‘a list.

```ocaml
SELECT x FROM x <= [1;1;2;2;3;3];
SELECT DISTINCT x FROM x <= [1;1;2;2;3;3];
SELECT y, x FROM (x, y) <= [("a", 1); ("b", 2)];
(SELECT (int * string) list = [((1, "a"), (2, "b"))]
(*) without FROM, WHERE, and HAVING *)
SELECT x, y;
SELECT x, y GROUP BY x;;
SELECT x, y ORDER BY x;;
(SELECT (count x) FROM x <= 1;;
```

Figure 15: Examples of the SELECT Clause

3 SEMANTICS AND COMPILATION

To provide a more reliable code transformation for the Select Expression, in this section, we formalise the semantics of SelectML.

Expressions

\[
e ::= \ldots | \{ e \} | \mathcal{P}
\]

Select Expressions

\[
q ::= \text{SELECT} | \text{SELECT DISTINCT}
\]

Select Clauses

\[
l ::= e \text{ o}
\]

From Clauses

\[
f ::= e \text{ FROM } f
\]

Source Patterns

\[
\chi ::= x | \{ x_1, \ldots, x_n \}
\]

Source Expressions

\[
s ::= \chi \leftrightarrow e | s, s
\]

Where Clauses

\[
w ::= f | f \text{ WHERE } e
\]

Group-by Clauses

\[
g ::= w | \text{GROUP } b\text{ USING } e
\]

Order-by Clauses

\[
o ::= g | g \text{ ORDER } b\text{ USING } e
\]

Query Plans

\[
P ::= \mathcal{E} | D_{x}(e) | \sigma_{e}(\mathcal{P}) | \Pi_{e}(\mathcal{P}) | \mathcal{P} \times \mathcal{P} | \epsilon_{G}(\mathcal{P}) | \epsilon_{Sk}(\mathcal{P}) | \epsilon_{U}(\mathcal{P})
\]

Figure 16: A Formal Syntax

Comparing to Figure 8, a more formal syntax is given in Figure 16. Here, the HAVING clause is left out for simplicity. Because it produces a selection plan \(\sigma\) just as the WHERE clause, only that it is handled after the GROUP BY clause and before the ORDER BY clause. For the ORDER BY clause, only USING e is preserved, as it is sufficient to express the semantics of ASC and DESC.

Source Patterns \(\chi\) can either be a variable \(x\), or a tuple of variables \((x_1, \ldots, x_n)\). Since variables \(x\) are also valid expressions, the symbol \(\chi\) is to denote expressions which consist of variables in following sections. Also for expediency, the operator ++ is used to indicate the concatenation of two tuples:

\[
(\chi_1, \ldots, \chi_n) \text{ ++ } (\chi_{n+1}, \ldots, \chi_m) = (\chi_1, \ldots, \chi_m)
\]

Query plans \(\mathcal{P}\) are a group of transient expressions, which will be explained in Section 3.2. A formal semantics of SelectML is provided in Section 3.3. And the translation schema from query plans to plan-free OCaml expressions is given in Section 3.4.

3.1 An Example

We reuse the example in Figure 2 to explain the handling of a Select Expression, shown in Figure 17. Aggregation functions firstrow and sum are predefined in Stdbib, of the types (‘a, ‘a) agg and (float, float) agg respectively. Function firstrow extracts the first row from a group of identical rows, while function sum returns the sum of a group of floating numbers. The Select Expression is firstly type-checked to determine the output type, which is (string * float) list in this example.

```ocaml
1 type order = ( id: int; price: float; month: string )
2 SELECT a.month, (sum a price) FROM a <- orders
3 WHERE a.month >= "2021-01" && a.month <= "2021-12"
4 GROUP BY a.month ORDER BY (sum a price) DESC
```

Figure 17: The Example of Select Expression

The generated query plan will be eventually translated to OCaml expressions which is plan-free for real execution (see Section 3.4). The code in Figure 18 shows a possible translation making use of the primitives defined in module SelectML. The generated plan-free OCaml expressions are guaranteed to be sound in type, as type errors are already handled when typing the Select Expression.
3.2 Query Plans

The query plan is widely used in database implementations as a way to describe the execution flow of Select Queries. In SelectML, query plans are a group of transient expressions that are the output of the planning phase and will be eventually translated to plan-free OCaml expressions.

\[
\begin{align*}
E & \quad \text{Empty Source} \\
\mathcal{D}_\mathcal{P}(e) & \quad \text{Data Source} \\
\mathcal{P}_1 \times \mathcal{P}_2 & \quad \text{Cartesian Product} \\
\mathcal{U}(\mathcal{P}) & \quad \text{Deduplication}
\end{align*}
\]

\[
\sigma_\mathcal{E}(\mathcal{P}) \quad \text{Selection} \\
\Pi_\mathcal{E}/\chi(\mathcal{P}) \quad \text{Projection} \\
e_{\mathcal{G}_{\mathcal{E}/\chi}}(\mathcal{P}) \quad \text{Aggregation} \\
e_{\mathcal{S}_{\mathcal{E}}}(\mathcal{P}) \quad \text{Sorting}
\]

Figure 19: Query Plans

A query plan denotes an abstract operation that will be performed on the input data. The query plans listed in Figure 19 can be categorized into three groups by their functionalities:

1) Providing data to outer query plans:
   - \( E \) provides the empty data source.
   - \( \mathcal{D}_\mathcal{P}(e) \) turns the expression \( e \) into a data source and rename the row of \( e \) to \( \chi \). That said, the variables of \( e \) can be referred as \( \chi \) in outer query plans.
   - \( \mathcal{P}_1 \times \mathcal{P}_2 \) returns the Cartesian product of plans \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \), and rename the new data source to \( \phi(\mathcal{P}_1)++\phi(\mathcal{P}_2) \).

\( \phi(\mathcal{P}) \) is defined as the name function for plan \( \mathcal{P} \). It returns a list of names to which the columns have been renamed:

\[
\begin{align*}
\phi(E) & = () \\
\phi(\mathcal{D}_\mathcal{P}(e)) & = \chi \\
\phi(\mathcal{P}_1 \times \mathcal{P}_2) & = \phi(\mathcal{P}_1)++\phi(\mathcal{P}_2) \\
\phi(\mathcal{U}(\mathcal{P})) & = \phi(\mathcal{P})
\end{align*}
\]

2) Performing actions to rows:
   - \( \sigma_\mathcal{E}(\mathcal{P}) \) performs a selection on rows that evaluate to \( \text{true} \) under the predicate \( e \) from the inner plan \( \mathcal{P} \).
   - \( e_{\mathcal{S}_{\mathcal{E}}}(\mathcal{P}) \) sorts the rows using the compare function \( e_\chi \) on the key \( e \) from the plan \( \mathcal{P} \).
   - \( \mathcal{U}(\mathcal{P}) \) removes duplicate rows from the plan \( \mathcal{P} \).

3) Performing actions to columns:
   - \( \Pi_\mathcal{E}/\chi(\mathcal{P}) \) projects the rows from \( \mathcal{P} \) to the expression \( e \), and rename the columns to \( \chi \).

- \( e_{\mathcal{G}_{\mathcal{E}/\chi}}(\mathcal{P}) \) groups the rows from \( \mathcal{P} \) on the key \( e \), then performs aggregations on the groups using aggregate functions \( e_\alpha \), and finally rename the aggregated columns to \( \chi \).

3.3 Typing and Planning

Queries can be statically typed in database PostgreSQL, where the types of columns in the Select Query are determined ahead of the execution. Operations like comparing columns with distinct types are not allowed, e.g. \( x=a \) and \( x=b \) reports error for column \( a \) having type int and \( b \) having type varchar. Queries can also be dynamically typed in MySQL and SQLite, where the types are checked at runtime. SelectML is statically typed along with the Select Expression. We present the operational semantics here to describe the typing and planning rules for the Select Expression, shown in Figure 20.

![Figure 18: The Example of Translated Code](image)

![Figure 20: Typing and Planning](image)
where $\tau$ ranges over OCaml types. The judgement $\vdash \chi : \tau \Rightarrow \Delta$ indicates the variable bindings created by source pattern $\chi$ of type $\tau$ are described by $\Delta$. A variable yields a single binding (rule VAR), while a tuple yields the union of its bindings (rule TUPLE). Operator is used to indicate the union of $\Delta_1$ and $\Delta_2$.

The judgement $\Gamma \vdash e : \tau \Rightarrow e'$ indicates that in environment $\Gamma$ the expression $e$ of type $\tau$ is transformed to $e'$. The aggregation $\{e_1, e_2\}$ of type $\tau_2$ is transformed to $\{e'_1, e'_2\}$ (rule AGGREGATE). If there is no Select Expression present, the judgement $\Gamma \vdash e : \tau \Rightarrow e$ holds as no transformation is performed.

The judgement $\Gamma \vdash \bullet \Rightarrow \mathcal{P}, \Delta$ indicates that in environment $\Gamma$ the plan generated for clause $\bullet$ is $\mathcal{P}$, and the bindings created by $\bullet$ are described by $\Delta$. The FROM clause can either generate a data source $\mathcal{D}_\gamma(e)$ and yield the bindings of the source patterns (rule SOURCE) or make a cartesian product of two data sources and combine their bindings (rule CARTESIAN-PRODUCT).

The Select Expression expects its subclauses to have type $\sigma$, so the type of the Select Expression becomes $\mathbf{list}$ (rule SELECT-DISTINCT).

The WHERE clause requires that the predicate $e$ has type bool, and produce a selection $\mathcal{S}_\epsilon(P)$ (rule WHERE). A single SELECT clause produces a single projection $\Pi_{\epsilon}(P)$ (rule SELECT-ONLY). A SELECT clause together with an ORDER BY clause yield a $\mathbb{Π}$--$\mathbb{Π}$ sequence (rule ORDER-By). A SELECT together with a GROUP BY yield a $\mathbb{Π}$--$\mathbb{Π}$ sequence (rule GROUP-By). And when all three clauses are present, we will get a $\mathbb{Π}$--$\mathbb{Π}$--$\mathbb{Π}$--$\mathbb{Π}$ sequence.

### 3.3.3 Auxiliary Functions

Auxiliary functions simplify the rules. Function $\text{expand-product}(e) \Rightarrow \chi, \sigma$ expands expression $e$ of a product type (tuples, records, etc.) into 2 parts: i) a tuple of sub-expressions $e_\sigma$; and ii) a pattern $\chi$ used to restore the original expression. For instance, the tuple $(x+y, x-y)$ can be expanded to a tuple of sub-expressions $(x+y, x-y)$ and the pattern $(c_1, c_2)$. Similarly, the record $(a = x+y; b = x-y)$ can be expanded to tuple $(x+y, x-y)$ and pattern $(a = c_1; b = c_2)$. The fresh variables $c_1, c_2$ function as the references for restoring the original expression. That is to say, plan $\Pi_{(x+y, x-y)}(P)$ is equivalent to $\Pi_{(c_1,c_2)}(\Pi_{(x+y, x-y)}(P))$ in terms of their computing results.

Code

```plaintext
SELECT x+y, x-y FROM ... ORDER BY x-y
```

will generate the plan $\Pi_{(x+y, x-y)}(\Pi_{(x+y, x-y)}(P))$. Function $\text{expand-aggregate}(e_1, \ldots, e_g) \Rightarrow (e'_1, \ldots, e'_g)$, $\chi_0, \chi_a, e_\sigma, e_\tau$ takes a list of expressions $(e_1, \ldots, e_g)$ as the input, where $e_g$ serves as the group key, and $(e_1, \ldots)$ are the expressions in the SELECT and ORDER BY clauses. Similar to $\text{expand-product}$, the outputs are: i) a tuple of sub-expressions $e_\sigma$ which are aggregation-free; ii) a pattern $\chi_0$ that is used to rename plan $\Pi_{e_\sigma}(P)$; iii) a tuple of extracted aggregate functions $e_\sigma$; iv) a pattern $\chi_a$ to rename plan $\mathcal{G}_{e_\sigma}(P)$; v) an aggregation-free expression $e_\tau$ that serves as the group key; and vi) a list of expressions $(e'_1, \ldots)$ that will restore the original expressions $(e_1, \ldots)$. For instance, if we let $e_1 = (\text{count}(x+y)+2$ and $e_2 = y$, then $\text{expand-aggregate}(e_1, e_2)$ will result in, $e'_1 = c_3 + 2, \chi_0 = c_2, \chi_0 = (c_3, c_4), e_0 = (\text{count}, \text{firstrow}), \chi_a = (c_1, c_2), e_\tau = (x+y, y)$. Code

```plaintext
SELECT (\text{count}(x+y)+2, y FROM ... GROUP BY y
```

will generate $\Pi_{(c_3, c_4)}(\Pi_{(x+y, y)}(P))$. When $e_\tau$ is not given, the function regresses to $\text{expand-aggregate}(e_1, \ldots) \Rightarrow (e'_1, \ldots, \chi_0, e_\sigma, e_\tau$.

#### 3.3.2 Special Cases

The Select Expression does not always produce a list when there is exactly one row in the result. As shown in Figure 21. Under any of the following circumstances:

1. the FROM clause is absent (Figure 22); or
2. there exists aggregation but the GROUP BY is absent (Figure 23), with the absence of the WHERE clause and HAVING clause, the result data will be a single value instead of a list. The sorting plan $\mathcal{S}$ becomes useless in such scenario, so it can be safely removed.

![Figure 21: Special Cases of Select Expression](image)

![Figure 22: No FROM clause](image)

![Figure 23: No GROUP BY clause](image)

### 3.4 Translation Schema

The semantics of the query plans mentioned in Section 3.2 can be captured by the module SelectML, whose signature is listed in Figure 24. type 'a t stands for the computation medium that will be passed among primitives, while type 'a src controls the input and output data type of the Select Expression. input and output are used for transformation between 'a t and 'a src. Although the type src and t are both defined as List, the types and primitives can be changed to serve other data types (see Section 4).
The translation schema is given in Figure 25. The generated OCaml expressions are type-safe since the expressions passed to the primitives are already type-checked when generating the plans.

![Figure 24: Signature of module SelectML](Image)

![Figure 25: Translation Schema](Image)

- For sorting \( eS_e(P) \), \( cmp \) and \( key \) are created for comparison on the sorting key \( e \), where \( cmp \) is a C-style compare function (cf. line 7 in Figure 24). Then \( SelectML.sort \) is called in \( P \).
- For deduplication \( U(P) \), \( SelectML.unique \) is called to deduplicate the rows.
- For grouping \( eG_{e_1,...,e_n}(P) \), \( SelectML.group \) is used to group the rows against the key \( e \), and perform aggregation to the groups with aggregate functions \( e_1, \ldots, e_n \). The type of grouping key \( e \) corresponds to \( 'c \) on line 10 in Figure 24.
- For grouping \( G_{e_1,...,e_n}(P) \), \( SelectML.group_all \) is used to aggregate all rows as one group.
- Function \( combine(e_1, \ldots, e_n) \) is used to combine several aggregate functions as a single aggregate function. If the inputs for \( combine \) are of types \( ('a_1, 'b_1) agg, \ldots, ('a_n, 'b_n) agg \), the result will have type \( ('a_1, \ldots, 'a_n), ('b_1, \ldots, 'b_n) agg \). As data supplied to \( SelectML.group \) is of type \( ('a_1, \ldots, 'a_n) \), it would be sound to serve the combined aggregate function as the argument.
- After we have translated the outermost plan, we have to cast the result back to type \( SelectML.src \) by calling \( SelectML.output \).
- When the result is bound to have exactly one row, \( SelectML.output \) is called instead of \( SelectML.src \) to cast the result into a single value rather than to \( SelectML.src \).

4 IMPLEMENTATION

This section will further elaborate the implementation of SelectML. The source code of SelectML is available online [6], which is developed based on a fork of the original OCaml compiler [1].

As described in 1.2, the OCaml compiler starts with the parsing phase (see file parsing/parse.ml), where the source program is read from a text file or an input channel (e.g. stdin), and parsed into a ParseTree (defined in parsing/parse.mli).

### 4.1 Typing the Select Expression

SelectML is implemented by modifying the parsing and typing phases of the OCaml compiler front-end as depicted in Section 1.2 and Figure 26. The Select Expression \( P.t.exp \) and aggregation \( Psamp.aggregte \) are added as new variants to ParseTree expression.

![Figure 26: Typing the Select Expression (Pt and Tt stands for ParseTree and TypedTree respectively)](Image)
With the optimisation on removing the unnecessary operations, the
plan \( \Pi_{\mathcal{G}_{\mathcal{S} \mathcal{E}}}(\Pi_{\mathcal{G}}(\mathcal{D}(t))) \) can be transformed to \( \mathcal{G}_{\mathcal{S} \mathcal{E}}(\mathcal{D}(t)) \). However, there is still superfluous operations in the result (line 3 to 4), which requires changes in the translation schema for the recognition of such cases. side effects of the Select Expression. The identical columns in the SELECT, GROUP BY, HAVING, ORDER BY clauses are merged into a unique one to avoid duplicate computations.

### 4.2 Flexibility of the Usage

As explained in the previous section, the implementation leaves much flexibility to the usage of the Select Expression since the module SelectML can be changed by the programmer.

#### 4.2.1 Shadowing the Primitives.

In OCaml, we can shadow a name by introducing either a local binding or a toplevel binding. For instance, redefining function group for a customised grouping behaviour, where only adjacent rows with the same group key will be grouped together.

```ocaml
1. let module SelectML = struct (* Local binding *)
2.  include SelectML
3.  let group key aggf ( =
4.  let cmp f a b = compare (f a) (f b) in
5.  let Agg (init, update, final) = aggf in
6.  1 |> List.to_seq
7.  |> Seq.group (fun a b -> cmp key a b = 0)
8.  |> Seq.map (fun g -> final (Seq.fold_left update init g))
9.  |> List.of_seq
end in
10. SELECT x FROM x <= [1;2;3;1;2;3] GROUP BY x ORDER BY x;
11. a test = : int list = [1; 1; 2; 2; 3] 3]

Figure 29: Shadow by Local Bindings
```

In Figure 29, the Select Expression on the line 11 produces different results from Figure 30. Line 11 is under the shadow of local module binding, and the rows with the same group key are not grouped together as they are not adjacent. Figure 30 produces a common result since the original module SelectML is used. By constrain, in Figure 31, the output for line 5 and line 8 are the same.

```ocaml
1. SELECT x FROM x <= [1;2;3;1;2;3] GROUP BY x ORDER BY x;
2. a test = : int list = [1; 1; 2; 2; 3] 3]

Figure 31: Shadow by Toplevel Bindings
```

#### 4.2.2 Changing the Input and Output Type.

Since module signature SelectMLType only defines abstract types and values, it is possible to change type 'a src and 'a t to any other data types. type 'a src controls the input and output type of the Select Expression. It can be changed to 'a array, 'a Seq. t or any user-defined data types as long as corresponding primitives input and output are provided. An example for array is given in Figure 32. The toplevel module SelectML conforms to module signature SelectMLType, with type src changed to array and primitives input, output defined for type casting between list and Seq. t.

Now the Select Expression can be used to handle arrays.
Figure 32: Changing the Input and Output Type

4.2.3 Changing the Intermediate Type. The module `SelectML` in `Stdlib` (file `stdlib/selectML.ml`) provides a simple implementation of the primitives of `SelectMLType` with the intermediate type being `list`.

Figure 33: Stdlib Module `SelectML`

The default implementation is listed in Figure 33, with the intermediate type `t` defined to be `list`. As the primitives accept and return data of type `t`, it is possible to change the intermediate type into any other data types that may bring benefits. Figure 34 is an example of changing the type to `array`. A great distinction lies in the implementation of function group, where Figure 33 adopts a sorting aggregate algorithm, and Figure 34 adopts a hash aggregate algorithm. It shows the flexibility of customising the behaviour of the Select Expression according to properties of the container types.

In mainstream databases, it is common to process data in batches, and a table can either be persisted or be a temporary existence in memory. We can let `type 'a src = 'a table` and `type 'a t = 'a chunk`, where table is defined to be the persistent or temporary data, and chunk implemented as a block linked list. Then the primitives can be implemented using parallel algorithms (Figure 35), since a block is processed as a batch while different list nodes can be processed simultaneously.

4.2.4 Generalisation. In Haskell, type classes are used for generalising the operation for a group of types. The instances of a type classes are passed implicitly to the callee functions. Since the list type is an instance of monad in Haskell, the list comprehension can be generalised to monad comprehensions [14] [3].

While there is no native support for ad-hoc polymorphism like type classes in OCaml, we have to make use of the module language to generalise the type of the Select Expression. The module signature `SelectMLType` can serve as the annotated type of a module argument. Implementations of `SelectMLType` can be passed as
arguments to shadow the original module SelectML, as we do it for function f in Figure 36.

```ocaml
1 (* invalid in OCaml *)
2 let f (module SelectML : SelectMLType) xs ys =
3   SELECT x, y FROM x <- xs, y <- ys WHERE x < y;;
```

**Figure 36: Generalising the Type and Operations**

However, the code in Figure 36 cannot typecheck since OCaml does not support higher kinded types for type parameters. Instead, we generalise function f by wrapping it inside a function as shown in Figure 37.

```ocaml
1 module F (SelectML : SelectMLType) = struct
2   let run xs ys = SELECT x, y FROM x <- xs, y <- ys WHERE x < y;;
3 end;
4
5 module F : ('a REPL output ->
6   functor (SelectML : SelectMLType) ->
7     sig
8       val run : ('a SelectML src -> 'a SelectML src ->
9         ('a x 'a) SelectML src
10     end
11
dis
12 (* Supplying primitives with the list implementation *)
13 let open F (ListImpl) in run ...
14
15 (* Supplying primitives with the array implementation *)
16 let open F (ArrayImpl) in run ...
```

**Figure 37: Generalisation with Functors**

The generalisation of the Select Expression allows the programmers to reuse the same business logics on various kinds of data types. Nevertheless, for the same data type, the user may supply modules with the different underlying implementations for the primitives to achieve distinct behaviours from the same logic (line 12 to 16 in Figure 37).

5 RELATED WORK

The typical list comprehension is in the form of \[ \{ e | (var \leftarrow \text{source}), \text{conditions} \} \], which corresponds to the `SELECT FROM WHERE` query in SQL. The generalised list comprehension [17] further supports `GROUP BY` and `ORDER BY`, available as an extension to [2] to GHC compiler. Just like List comprehension, a similar operation can be applied to other monad types (e.g., `Maybe` type) by the monad comprehension [3, 14].

As OCaml does not have native support for ad-hoc polymorphism and higher kinded types, the generalisation for types in SelectML have to be achieved using the module language [19] of OCaml as described in Section 4.2.4. The modular implicits [22] provides a solution for ad-hoc polymorphism in OCaml. With modular implicits, functions can accept implicit module arguments and suitable module arguments can be passed implicitly. It may benefits the implementation of SelectML by replacing the explicit shadowing of module SelectML with implicit bindings.

A denotational semantics and validation of SQL queries are given in the work [16]. Similar to this work, an operational semantics for the Select Expression in SelectML is presented in Section 3.

SQLite [9] is a lightweight SQL database, which can be embedded into softwares as program libraries. PostgreSQL [21] [5] is an open-source object-relational database with good support for user-defined functions and data types. It also support JIT execution for SQL queries. SingleStore [7] (formerly known as MemSQL) is a distributed SQL HTAP (Hybrid Transactional and Analytical Processing) database that features in-memory storage and JIT compilation for the speed of data processing. HyPer [18] is another in-memory HTAP database that utilises JIT for high performance execution. Hadoop [20] is a distributed file system designed for efficiently processing and storing big data. MapReduce [12] is a programming model and an associated implementation built on top of the Hadoop system for processing and generating large data sets. Hive [11] is a data warehouse on Hadoop which provides an SQL-like query language for data processing and analysis. Spark [24] is Scala framework which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. The work [13] talks about the main coordinates for data processing and SQL features for data analysis.

Rule-based optimisation [8] is an optimisation technique that rewrites the query plan using a set of logical rules to a better plan that may reduce the amount of computations. Cost-based optimisation [15] is a more powerful optimisation technique that finds the best query plans by computing the minimum estimated cost of all possible plans. OptGen [10] is a tool for generating verified local optimisations from optimisation rules. It can be used to implement the cost-based optimisation for databases.

6 CONCLUSION AND FUTURE WORK

SQL being widely adopted in mainstream databases, the Select Query is the most commonly operation in SQL. The Select Query handles data in functional manner as it does not mutate the input and can be used as pure functions. Its conciseness of describing the expected result set makes the Select Query a good complementary for programming languages like OCaml.

In this work, SelectML as an SQL frontend on top of OCaml, is presented for the purpose of data analysis. Two new constructs have been added to OCaml expressions: the Select Expression, and the aggregation. The new constructs can be used as other OCaml expressions seamlessly. To model the behaviour of SelectML, an operational semantics is presented to explain the typing and planning rules for the Select Expression and the aggregation. The typing rules specify how the Select Expression conforms with the OCaml type system, and the planning rules imply what kind of plans will be generated for a given Select Expression. A translation schema from query plans to plan-free OCaml expressions is also provided.

The semantics that have been proposed can also serve as a reference implementation for SQL dialects. SelectML is implemented as a language extension to OCaml, and project is available online. The implementation makes the Select Expression a flexible construct for programmers by giving them great freedom to customise the underlying operation that is suitable for their use case. The input and output type, the intermediate type, and the primitives for module SelectML can all be changed to user-defined versions. However, the project is still a prototype and subjects to several limitations. The join operations, window functions, indexes and optimisations are important features for SQL database that will be addressed in the future.
REFERENCES
[19] An SQL frontend on top of OCaml for Data Analysis IFL'34, September 2022, Danmark

A LIMITATIONS AND FUTURE WORKS

Other missing features include specialised join syntax (Section A.1), window functions (Section A.2), and plan optimisations A.3.

A.1 Joins

Currently, the Cartesian product $P \times P$ together with selection $\sigma$ are used to simulate the inner join. In this way, all possible combinations of elements from the two data sources have to be computed and stored in memory before performing the selection. Although the memory consumption can be alleviated by using lazy data structures like Seq, it still leads to mediocre performance in most cases.

![Figure 38: Join Operations](image)

There are more types of join operations apart from inner joins, like natural joins and outer joins, shown in Figure 38. Users generally don’t have to write the join condition using syntax JOIN ON explicitly. For queries like FROM `xs`, `ys` WHERE `x` = `y`, the condition residing in the WHERE clause is extracted, and if the condition can be used as a join condition, the Cartesian product will be replaced with an inner join. The performance can be significantly improved since inner join can be implemented with merge join or hash join, rather than nested loops for Cartesian product.

![Figure 39: Primitives for Join Operations](image)

As for future work, a new plan $P_1 \leftarrow \pi_{x \neq y} P_2$ will be added for the inner join. And new primitives for implementing the inner join (Figure 39).

![Figure 40: Using Join Operations](image)

When the hash keys of both sides can be determined (Figure 40), plan $\pi_{x \neq y}$ can be translated to calls for join_eq, which should be implemented as hash join or some other efficient algorithm. Otherwise, join is used to perform a regular join with conditions, and join is supposed to filter those elements at the time of computing the product.
A.1.1 Outer Joins. The example in Figure 41 shows the result of left outer join in PostgreSQL 14.2. The outer join of two data sources is like the inner join plus the remaining data from any of the two sides. Left outer joins preserve the data from the left side, right outer joins preserve the data from the right side, and full outer joins preserve both sides. Empty cells will be filled with NULLs. In SelectML, to support such feature, we have to replace NULL with None, the column type being changed from 'a' to 'a' option. Similarly, examples of right and full outer joins are given in Figure 42.

A.2 Window Functions

In SQL, window functions are a group of functions that compute the result for each row using the values of one or more other rows, which is useful for analytical data processing. For instance, to compute the average of 15 most recent records/rows for financial data. The PARTITION BY clause creates window frames for the rows with the same value on x, and these rows will be used as the input for aggregate function COUNT.

As shown in the examples, the window function is an aggregate function plus a window specification. Since it is possible for a window to slide during the execution (affected by ORDER BY, ROWS, RANGE), the aggregate function has to support appending a value to its tail, as well as removing a value from its head. Hence, the definition for aggregate functions should be changed to Figure 46.

It is easy to support the window function that produces the correct result, while the real challenge will be supporting it efficiently.
A.3 Optimisations

The same SQL query can be translated to several equivalent query plans, that may differ in time complexity and/or space consumptions. Therefore, it is important to find the best plan in terms of performance for industry level databases.

The query plan optimisation is the core part of the SQL optimiser in mainstream databases. Databases, as the data management system, have access to the meta information (which is called Schema) about the tables being queried, including the number of rows, the frequency of being queried. By accessing and maintaining these information each time a query happens, the databases system is able to perform a cost-based optimisation. For instance, the optimiser may decide an Aggregate operation is implemented in hash aggregation or ordered aggregation depending on the size of the input.

On the contrary, SelectML on top of OCaml, as a statically typed language, compiles programs to executables ahead-of-time. It does not have the runtime information for data that are given to the Select Expression, hence it is not able to perform cost-based optimisations for the query plans without modifying the runtime.

Nevertheless, it always possible to perform rule-based optimisations with knowledge from the compile time. Following are two possible rules for optimising plans,

- eliminating unnecessary plans, e.g. removing the identity projection: \(\Pi_{(c_1, c_2)}(\Pi_{(x,y)}/(c_1, c_2)(P)) \Longrightarrow \Pi_{(x,y)}(P)\),
- pushing down the selection in order to save the computation on joining:

\[\sigma_{x<1 \land y>1}(D_x(x) \times D_y(y)) \Longrightarrow \sigma_{x<1}(D_x(x)) \times \sigma_{y>1}(D_y(y))\]

A.3.1 Indexes. Indexes in databases are used to locate data quickly without scanning the whole table, and they are usually implemented in B-tree.

![Figure 47: Condition Involving Comparisons](image)

Indexes is an important optimisation point when there are comparisons in the WHERE condition (Figure 47). Without indexes on \(x\), plan \(\sigma_{x<1}(D_x(x))\) will have to perform a full scan on table \(xs\) to get the data satisfying \(x < 1\). With indexes created on \(x\), the full scan can be replaced with a range scan only on interval \((-\infty, 1)\). Also, the trivial nested loop join algorithm can be substituted by merge join and index lookup join if indexes are available.

This has posed an issue for SelectML, since it is not an easy task to check if there are some indexes defined on the input data within the type system of OCaml. For instance, describing the index created for the first column of \((\text{int} \times \text{int})\text{ list}\). And to deal with this, some efforts have to be made on the OCaml type system.

Figure 48 poses a possible solution for the type system. Indexes are modeled by module type \(\text{datatype ORDER BY fields}\). For tuple types like \((\text{int} \times \text{int})\text{ list}\) (line 16, 17), the \(\text{fields}\) are integers to denote which columns are to be indexed, i.e. \(0\) denotes the first column of \((\text{int} \times \text{int})\text{ list}\). For records like \(\text{order}\text{ list}\) (line 21, 22), the \(\text{fields}\) should be the field names of the record type.

![Figure 48: Typing Indexes](image)