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:
Yan Dong, Yahui Song, and Wei-Ngan Chin. 2022. An SQL Frontend on top of OCaml for Data Analysis. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'34).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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:

```
type order = { id: int; price: float; month: string }

let group f l =
    let eq a b = compare (f a) (f b) = 0 in
    List.to_seq l |> Seq.group eq
                  |> Seq.map List.of_seq |> List.of_seq

let income_by_month : order list -> (string * float) list =
    fun orders -> orders
    |> List.filter (fun o ->
           o.month >= "2021-01" && o.month <= "2021-12")
    |> List.sort (fun a b -> compare a.month b.month)
    |> group (fun o -> o.month)
    |> List.map (fun l ->
        List.fold_left
            (fun (m, income) o -> (m, o.price +. income))
            ((List.hd l).month, 0.0) l)
    |> 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

IFL'34, September 2022, Danmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

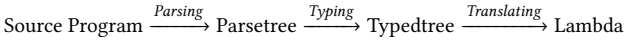


Figure 3: An Overview of OCaml Frontend

Figure 3 gives an overview of the compilation phases in the OCaml frontend. 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.

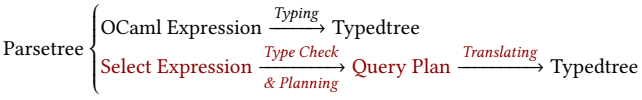


Figure 4: An Overview of *SelectML*

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 Typedtree. 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 retrofit 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.

```
(* as arguments *)
f 123 (SELECT x FROM x <- xs)

(* as return values *)
let g xs = SELECT x FROM x <- xs

(* as operands *)
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 */
SELECT ... WHERE EXISTS (SELECT ...)
SELECT ... WHERE col IN (SELECT ...)
```

```
(* SelectML *)
SELECT ... WHERE List.length (SELECT ...) > 0
SELECT ... WHERE List.mem col (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 an 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.

```
/* SQL */
WITH t AS (SELECT ...)
SELECT ...
```

```
(* SelectML *)
let t = SELECT ... in
SELECT ...
```

Figure 7: Common Table Expressions

2.2 An Informal Syntax

The syntax for *SelectML* are presented in Figure 8. *SelectML* expressions consist of regular OCaml expressions including constants, variables x, tuples $(e_1, \ldots, e_n)$, list, and functions. On top of OCaml expressions, there are two new syntactic constructs: 1) The Select Expression q; 2) The aggregation $\{e_1\ e_2\}$ where e_1 is the aggregate function, and e_2 is the argument.

Expressions	e	::=	$\ldots \mid q \mid \{e\ e\}$
Select Expressions	q	::=	SELECT [DISTINCT] e [FROM s] [WHERE e] [GROUP BY e] [HAVING e] [ORDER BY o]
Source Expressions	s	::=	$x \leftarrow e \mid (x_1, \ldots, x_n) \leftarrow e \mid s, s$
Order Expressions	o	::=	e [ASC | DESC | USING e] $\mid o, o$

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 $x \leftarrow$ *source*, which makes it easier to resolve parser conflicts. x in source expressions is a shorthand for $x \leftarrow x$. The type of *source* is expected to be 'a **list**. A direct comparison of the SQL syntax and *SelectML* is given in Figure 9.

```
/* SQL */
SELECT x, y FROM xs AS x, ys AS y;
```

```
(* SelectML *)
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.

```
SELECT x FROM x <- xs WHERE f x;;
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.

```
/* SQL */
SELECT x, COUNT(y) FROM t GROUP BY x;
```

```
(* SelectML *)
SELECT x, {count y} FROM (x, y) <- 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 $\{e_1\ e_2\}$ is used to denote the application of aggregate functions, where e_1 is the aggregate function, and e_2 is the argument column. Some databases (e.g., PostgreSQL) support user-defined aggregate functions, where users may supply the initial state, the transition function, and the function to get the final result.

```
type ('a, 'b, 'c) aggfunc = 'c * ('c -> 'a -> 'c) * ('c -> 'b)
type (_, _) agg = Agg : ('a, 'b, 'c) aggfunc -> ('a, 'b) agg

val firstrow : ('a, 'a) agg
val agg_min : ('a, 'a) agg
val agg_max : ('a, 'a) agg
val count : ('a, int) agg
val sum : (float, float) agg
val avg : (float, float) agg
```

Figure 12: Aggregate Functions

```
val mkagg : 'c -> ('c -> 'a -> 'c) -> ('c -> 'b) -> ('a, 'b) agg

(* create an aggregate function *)
let count = mkagg 0 (fun n _ -> n + 1) (fun n -> n);;

(* usage inside Select Expression *)
SELECT {count x} FROM x <- [1;2;3];;

(* usage outside Select Expression *)
let Agg (init, update, final) = count in
final (List.fold_left update init [1;2;3])

(* invalid usage *)
let _ = {count [1;2;3]};;
Error: Standalone aggregate is not allowed
```

Figure 13: Usages of Aggregate Functions

SelectML supports user-defined aggregate functions by default. As shown in Figure 12, predefined aggregate functions (i.e. agg_min,

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 (`init`), the transition function (`update`), and the result function (`final`). Usage of the syntax $\{e_1\ e_2\}$ 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` e for using e 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.

```
1  let odd_first a b =
2      let x = a mod 2 = 0 in
3      let y = b mod 2 = 0 in
4      compare x y;;
5
6  SELECT x, y
7  FROM (x, y) <- [("a", 2); ("a", 3); ("b", 4); ("b", 5)]
8  ORDER BY x ASC, y USING odd_first;;
9
10 -:(string * int) SelectML.src=[("a",3); ("a",2);("b",5);("b",4)]
```

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`.

```
1  SELECT x FROM x <- [1;1;2;2;3;3];;
2  - : int list = [1; 1; 2; 2; 3; 3]
3
4  SELECT DISTINCT x FROM x <- [1;1;2;2;3;3];;
5  - : int list = [1; 2; 3]
6
7  SELECT y, x FROM (x, y) <- [("a", 1); ("b", 2)];;
8  - : (int * string) list = [(1, "a"); (2, "b")]
9
10 (* without FROM, WHERE, and HAVING *)
11 SELECT x, y;;
12 SELECT x, y GROUP BY z;;
13 SELECT x, y ORDER BY z;;
14
15 (* aggregation without GROUP BY, WHERE, and HAVING *)
16 SELECT {count x} FROM x <- t;;
```

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*.

$$
\begin{array}{rrcl}
\text{Expressions} & e & ::= & \ldots \mid q \mid \{e\ e\} \mid \mathcal{P} \\
\text{Select Expressions} & q & ::= & \texttt{SELECT}\ l \mid \texttt{SELECT DISTINCT}\ l \\
\text{Select Clauses} & l & ::= & e\ o \\
\text{From Clauses} & f & ::= & \epsilon \mid \texttt{FROM}\ s \\
\text{Source Patterns} & \chi & ::= & x \mid (x_1, \ldots, x_n) \\
\text{Source Expressions} & s & ::= & \chi \leftarrow e \mid s, s \\
\text{Where Clauses} & w & ::= & f \mid f\ \texttt{WHERE}\ e \\
\text{Group-by Clauses} & g & ::= & w \mid w\ \texttt{GROUP BY}\ e \\
\text{Order-by Clauses} & o & ::= & g \mid g\ \texttt{ORDER BY}\ e\ \texttt{USING}\ e \\
\text{Query Plans} & \mathcal{P} & ::= & \mathcal{E} \mid \mathcal{D}_{\chi}(e) \mid \sigma_{e}(\mathcal{P}) \mid \Pi_{e/\chi}(\mathcal{P}) \mid \\
 & & & \mathcal{P} \times \mathcal{P} \mid {}_{e}\mathcal{G}_{e}(\mathcal{P}) \mid {}_{e}\mathcal{S}_{e}(\mathcal{P}) \mid \mathcal{U}(\mathcal{P})
\end{array}
$$

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 σ 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 χ 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 χ 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:

$$
\begin{array}{rcl}
(\chi_1, \ldots, \chi_n)\ \texttt{++}\ (\chi_{n+1}, \ldots, \chi_m) & = & (\chi_1, \ldots, \chi_m) \\
(e_1, \ldots, e_n)\ \texttt{++}\ (e_{n+1}, \ldots, e_m) & = & (e_1, \ldots, e_m)
\end{array}
$$

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 **`Stdlib`**, 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.

```
1 type order = { id: int; price: float; month: string }
2
3 SELECT o.month, {sum o.price} FROM o <- orders
4 WHERE o.month >= "2021-01" && o.month <= "2021-12"
5 GROUP BY o.month ORDER BY {sum o.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.

```
orders
|> SelectML.input
|> SelectML.filter (fun o ->
                o.month >= "2021-01" && o.month <= "2021-12")
|> SelectML.map (fun o -> o.month, o.price)
|> SelectML.group (fun (__col_1, __col_2) -> __col_1)
    (let Agg (init1, update1, final1) = Stdlib.firstrow in
     let Agg (init2, update2, final2) = Stdlib.sum in
     Agg ((init1, init2),
       (fun (acc1, acc2) (x1, x2) ->
           update1 acc1 x1, update2 acc2 x2),
       (fun (acc1, acc2) -> final1 acc1, final2 acc2)))
|> let key (__col_1, __col_3) -> __col_3) in
   SelectML.sort (fun a b -> compare (key b) (key a))
|> SelectML.output
```

Figure 18: The Example of Translated Code

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.

$\mathcal{E}$	Empty Source	$\sigma_e(\mathcal{P})$	Selection
$\mathcal{D}_\chi(e)$	Data Source	$\Pi_{e/\chi}(\mathcal{P})$	Projection
$\mathcal{P}_1 \times \mathcal{P}_2$	Cartesian Product	${}_e\mathcal{G}_{e_a/\chi}(\mathcal{P})$	Aggregation
$\mathcal{U}(\mathcal{P})$	Deduplication	${}_e\mathcal{S}_{e_f}(\mathcal{P})$	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:

- $\mathcal{E}$ provides the empty data source.
- $\mathcal{D}_\chi(e)$ turns the expression e into a data source and rename the row of e to χ. That said, the variables of e can be referred as χ 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:

$$\begin{array}{rcl|rcl} \phi(\mathcal{E}) & = & () & \phi(\sigma_e(\mathcal{P})) & = & \phi(\mathcal{P}) \\ \phi(\mathcal{D}_\chi(e)) & = & \chi & \phi(\Pi_{e/\chi}(\mathcal{P})) & = & \chi \\ \phi(\mathcal{P}_1 \times \mathcal{P}_2) & = & \phi(\mathcal{P}_1) \text{ ++ } \phi(\mathcal{P}_2) & \phi({}_e\mathcal{G}_{e_a/\chi}(\mathcal{P})) & = & \chi \\ \phi(\mathcal{U}(\mathcal{P})) & = & \phi(\mathcal{P}) & \phi({}_e\mathcal{S}_{e_f}(\mathcal{P})) & = & \phi(\mathcal{P}) \end{array}$$

2) Performing actions to rows:

- $\sigma_e(\mathcal{P})$ performs a selection on rows that evaluate to `true` under the predicate e from the inner plan $\mathcal{P}$.
- ${}_e\mathcal{S}_{e_f}(\mathcal{P})$ sorts the rows using the compare function e_f 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:

- $\Pi_{e/\chi}(\mathcal{P})$ projects the rows from $\mathcal{P}$ to the expression e, and rename the columns to χ.
- ${}_e\mathcal{G}_{e_a/\chi}(\mathcal{P})$ groups the rows from $\mathcal{P}$ on the key e, then performs aggregations on the groups using aggregate functions e_a, and finally rename the aggregated columns to χ.

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. a=b reports error for column a having type `int` and b having type `varchar`. Queries can also be dynamically type 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.

$$\boxed{\vdash \chi : \tau \Rightarrow \Delta}$$

$$\textsc{Var}\ \frac{}{\vdash x : \tau \Rightarrow \{x : \tau\}}$$

$$\textsc{Tuple}\ \frac{\vdash x_1 : \tau_1 \Rightarrow \Delta_1 \quad \ldots \quad \vdash x_n : \tau_n \Rightarrow \Delta_n}{\vdash (x_1, \ldots, x_n) : (\tau_1, \ldots, \tau_n) \Rightarrow \Delta_1 \uplus \ldots \uplus \Delta_n}$$

$$\boxed{\Gamma \vdash e : \tau \Rightarrow e'}$$

$$\textsc{Aggregate}\ \frac{\Gamma \vdash e_1 : (\tau_1, \tau_2)\ \text{agg} \Rightarrow e_1' \quad \Gamma \vdash e_2 : \tau_1 \Rightarrow e_2'}{\Gamma \vdash \{e_1\ e_2\} : \tau_2 \Rightarrow \{e_1'\ e_2'\}}$$

$$\textsc{Select}\ \frac{\Gamma \vdash l : \tau \Rightarrow \mathcal{P}, \_}{\Gamma \vdash \texttt{SELECT}\ l : \tau \Rightarrow \mathcal{P}}$$

$$\textsc{Select-Distinct}\ \frac{\Gamma \vdash l : \tau \Rightarrow \mathcal{P}, \_}{\Gamma \vdash \texttt{SELECT DISTINCT}\ l : \tau \Rightarrow \mathcal{U}(\mathcal{P})}$$

$$\boxed{\Gamma \vdash \bullet : \tau \Rightarrow \mathcal{P}, \Delta}$$

$$\textsc{From}\ \frac{\Gamma \vdash s : \tau \Rightarrow \mathcal{P}, \Delta}{\Gamma \vdash \texttt{FROM}\ s : \tau \Rightarrow \mathcal{P}, \Delta} \qquad \textsc{Source}\ \frac{\Gamma \vdash e : \tau\ \text{list} \Rightarrow e' \quad \vdash \chi : \tau \Rightarrow \Delta}{\Gamma \vdash \chi \leftarrow e : \tau \Rightarrow \mathcal{D}_\chi(e'), \Delta}$$

$$\textsc{Cartesian-Product}\ \frac{\Gamma \vdash s_1 : \tau_1 \Rightarrow \mathcal{P}_1, \Delta_1 \quad \Gamma \vdash s_2 : \tau_2 \Rightarrow \mathcal{P}_2, \Delta_2}{\Gamma \vdash s_1, s_2 : \tau_1 \text{++} \tau_2 \Rightarrow \mathcal{P}_1 \times \mathcal{P}_2, (\Delta_1 \uplus \Delta_2)}$$

$$\textsc{Where}\ \frac{\Gamma \vdash f : \tau \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e : \text{bool} \Rightarrow e'}{\Gamma \vdash f\ \texttt{WHERE}\ e : \tau \Rightarrow \sigma_{e'}(\mathcal{P}), \Delta}$$

$$\textsc{Select-Only}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e : \tau \Rightarrow e'}{\Gamma \vdash e\ w : \tau\ \text{list} \Rightarrow \Pi_{e'}(\mathcal{P}), \Delta}$$

$$\textsc{Order-By}\ \frac{\begin{array}{c}\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \Gamma, \Delta \vdash e_2 : \tau_2 \Rightarrow e_2' \\ \Gamma, \Delta \vdash e_f : \forall \alpha.\alpha \to \alpha \to \text{int} \Rightarrow e_f' \\ \Gamma, \Delta \vdash \textit{expand-product}(e_1', e_2') \Rightarrow (\chi_1, \chi_2), e_\pi\end{array}}{\Gamma \vdash e_1\ w\ \texttt{ORDER BY}\ e_2\ \texttt{USING}\ e_f : \tau_1\ \text{list} \Rightarrow \Pi_{\chi_1}({}_{\chi_2}\mathcal{S}_{e_f'}(\Pi_{e_\pi/(\chi_1 \text{++} \chi_2)}(\mathcal{P}))), \Delta}$$

$$\textsc{Group-By}\ \frac{\begin{array}{c}\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \Gamma, \Delta \vdash e_g : \tau_g \Rightarrow e_g' \\ \Gamma, \Delta \vdash \textit{expand-aggregate}(e_1', e_g') \Rightarrow e_1'', \chi_g, \chi_a, e_a, \chi_\pi, e_\pi\end{array}}{\Gamma \vdash e_1\ w\ \texttt{GROUP BY}\ e_g : \tau_1\ \text{list} \Rightarrow \Pi_{e_1''}({}_{\chi_g}\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{P}))), \Delta}$$

$$\textsc{Group-Order-By}\ \frac{\begin{array}{c}\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \Gamma, \Delta \vdash e_2 : \tau_2 \Rightarrow e_2' \\ \Gamma, \Delta \vdash e_g : \tau_g \Rightarrow e_g' \quad \Gamma, \Delta \vdash e_f : \forall \alpha.\alpha \to \alpha \to \text{int} \Rightarrow e_f' \\ \Gamma, \Delta \vdash \textit{expand-aggregate}(e_1', e_2', e_g') \Rightarrow (e_1'', e_2''), \chi_g, \chi_a, e_a, \chi_\pi, e_\pi \\ \Gamma, \Delta \vdash \textit{expand-product}(e_1'', e_2'') \Rightarrow (\chi_1, \chi_2), e_\pi'\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ w\ \texttt{GROUP BY}\ e_g\ \texttt{ORDER BY}\ e_2\ \texttt{USING}\ e_f : \tau_1\ \text{list} \Rightarrow \\ \Pi_{\chi_1}({}_{\chi_2}\mathcal{S}_{e_f'}(\Pi_{e_\pi'/(\chi_1 \text{++} \chi_2)}({}_{\chi_g}\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{P}))))), \Delta\end{array}}$$

Figure 20: Typing and Planning

We use Γ to denote the environment of variable bindings and Δ for the increment of bindings. A variable binding is like $\chi : \tau$,

where τ ranges over OCaml types. The judgement $\vdash \chi : \tau \Rightarrow \Delta$ indicates the variable bindings created by source pattern χ of type τ are described by Δ. 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 Δ_1 and Δ_2,

The judgement $\Gamma \vdash e : \tau \Rightarrow e'$ indicates that in environment Γ the expression e of type τ is transformed to e'. The aggregation $\{e_1\ e_2\}$ of type τ_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 Γ the plan generated for clause $\bullet$ is $\mathcal{P}$, and the bindings created by $\bullet$ are described by Δ. The `FROM` clause can either generate a data source $\mathcal{D}_\chi(e)$ 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 τ, so the type of the Select Expression becomes τ `list` (rule Select, Select-Distinct).

The `WHERE` clause requires that the predicate e has type `bool`, and produce a selection $\sigma_e(\mathcal{P})$ (rule Where). A single `SELECT` clause produces a single projection $\Pi_{e'}(\mathcal{P})$ (rule Select-Only). A `SELECT` clause together with an `ORDER BY` clause yield a $\Pi \rightarrow \mathcal{S} \rightarrow \Pi$ sequence (rule Order-By). A `SELECT` together with a `GROUP BY` yield a $\Pi \rightarrow \mathcal{G} \rightarrow \Pi$ sequence (rule Group-By). And when all three clauses are present, we will get a $\Pi \rightarrow \mathcal{G} \rightarrow \Pi \rightarrow \mathcal{S} \rightarrow \Pi$ sequence.

3.3.1 Auxiliary Functions. Auxiliary functions *expand-product* and *expand-aggregate* are introduced to simplify the rules.

Function *expand-product*$(e) \Rightarrow \chi, e_\pi$ expands expression e of a product type (tuples, records, etc.) into 2 parts: i) a tuple of sub-expressions e_π; and ii) a pattern χ 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 `(c1, c2)`. Similarly, the record `{a = x+y; b = x-y}` can be expanded to tuple `(x+y, x-y)` and pattern `{a = c1; b = c2}`. The fresh variables `c1`, `c2` function as the references for restoring the original expression. That is to say, plan $\Pi_{(\text{x+y, x-y})}(\mathcal{P})$ is equivalent to $\Pi_{(c_1,c_2)}(\Pi_{(\text{x+y, x-y})/(c_1,c_2)}(\mathcal{P}))$ in terms of their computing results. Code `SELECT x+y, x-y FROM ... ORDER BY x-y` will generate the plan $\Pi_{(c_1,c_2)}(_{c_2}\mathcal{S}_{\text{asc}}(\Pi_{(\text{x+y, x-y})/(c_1,c_2)}(\mathcal{P})))$.

Function *expand-aggregate*$(e_1, \ldots, e_g) \Rightarrow (e'_1, \ldots), \chi_g, \chi_a, e_a, \chi_\pi, e_\pi$ 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 *expand-product*, the outputs are: i) a tuple of sub-expressions e_π which are aggregation-free; ii) a pattern χ_π that is used to rename plan $\Pi_{e_\pi}(\mathcal{P})$; iii) a tuple of extracted aggregate functions e_a; iv) a pattern χ_a to rename plan $\mathcal{G}_{e_a}(\mathcal{P})$; v) an aggregation-free expression e_g 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 = `{count (x+1)} + 2` and e_g = `y`, *expand-aggregate*(e_1, e_g) will result in, $e'_1 = c_3 + 2$, $\chi_g = c_2$, $\chi_a = (c_3, c_4)$, e_a = `(count, firstrow)`, $\chi_\pi = (c_1, c_2)$, e_π = `(x+1, y)`. Code `SELECT {count(x+1)}+2, y FROM ... GROUP BY y` will generate $\Pi_{(c_3+2,c_4)}(_{c_2}\mathcal{G}_{(\text{count,firstrow})/(c_3,c_4)}(\Pi_{(\text{x+1,y})/(c_1,c_2)}(\mathcal{P})))$. When e_g is not given, the function regresses to *expand-aggregate*$(e_1, \ldots) \Rightarrow (e'_1, \ldots), \chi_a, e_a, \chi_\pi, e_\pi$.

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.

```
SELECT 1, 2;; (* no FROM clause *)
- : int * int = (1, 2)

(* aggregation without GROUP BY *)
SELECT {count x} FROM x <- [1;2;3];;
- : int = 3
```

Figure 21: Special Cases of Select Expression

$$\textsc{Empty}\ \frac{}{\Gamma \vdash \epsilon : \text{unit} \Rightarrow \mathcal{E}, ()}$$

$$\textsc{Select-Only'}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{E}, \Delta \qquad \Gamma, \Delta \vdash e : \tau \Rightarrow e'}{\Gamma \vdash e\ w : \tau \Rightarrow \Pi_{e'}(\mathcal{E}), \Delta}$$

$$\textsc{Order-By'}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{E}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e'_1 \quad \Gamma, \Delta \vdash e_2 : \tau_2 \Rightarrow e'_2 \quad \Gamma, \Delta \vdash \textit{expand-product}(e'_1, e'_2) \Rightarrow (\chi_1, \chi_2), e_\pi}{\Gamma \vdash e_1\ w\ \texttt{ORDER BY}\ e_2 : \tau_1 \Rightarrow \Pi_{\chi_1}(\Pi_{e_\pi/(\chi_1 +\!\!+ \chi_2)}(\mathcal{E})), \Delta}$$

$$\textsc{Group-By'}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{E}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e'_1 \quad \Gamma, \Delta \vdash e_g : \tau_g \Rightarrow e'_g \quad \Gamma, \Delta \vdash \textit{expand-aggregate}(e'_1, e'_g) \Rightarrow e''_1, \chi_g, \chi_a, e_a, \chi_\pi, e_\pi}{\Gamma \vdash e_1\ w\ \texttt{GROUP BY}\ e_g : \tau_1 \Rightarrow \Pi_{e''_1}(_{\chi_g}\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{E}))), \Delta}$$

$$\textsc{Group-Order-By'}\ \frac{\begin{array}{c}\Gamma \vdash w : \_ \Rightarrow \mathcal{E}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e'_1 \quad \Gamma, \Delta \vdash e_2 : \tau_2 \Rightarrow e'_2 \\ \Gamma, \Delta \vdash e_g : \tau_g \Rightarrow e'_g \qquad \Gamma, \Delta \vdash e_f : \forall \alpha.\alpha \rightarrow \alpha \rightarrow \text{int} \Rightarrow \_ \\ \Gamma, \Delta \vdash \textit{expand-aggregate}(e'_1, e'_2, e'_g) \Rightarrow (e''_1, e''_2), \chi_g, \chi_a, e_a, \chi_\pi, e_\pi \\ \Gamma, \Delta \vdash \textit{expand-product}(e''_1, e''_2) \Rightarrow (\chi_1, \chi_2), e'_\pi\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ w\ \texttt{GROUP BY}\ e_g\ \texttt{ORDER BY}\ e_2\ \texttt{USING}\ e_f : \tau_1 \Rightarrow \\ \Pi_{\chi_1}(\Pi_{e'_\pi/(\chi_1 +\!\!+ \chi_2)}(_{\chi_g}\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{E})))), \Delta\end{array}}$$

Figure 22: No `FROM` clause

$$\textsc{Select-Only''}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \qquad \Gamma, \Delta \vdash e : \tau \Rightarrow e' \qquad \Gamma, \Delta \vdash \textit{expand-aggregate}(e') \Rightarrow e'', \chi_a, e_a, \chi_p i, e_\pi}{\Gamma \vdash e\ w : \tau \Rightarrow \Pi_{e''}(\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{P}))), \Delta}$$

$$\textsc{Order-By''}\ \frac{\Gamma \vdash w : \_ \Rightarrow \mathcal{P}, \Delta \quad \Gamma, \Delta \vdash e_1 : \tau_1 \Rightarrow e'_1 \quad \Gamma, \Delta \vdash e_2 : \tau_2 \Rightarrow e'_2 \quad \Gamma, \Delta \vdash e_f : \forall \alpha.\alpha \rightarrow \alpha \rightarrow \text{int} \Rightarrow \_ \quad \Gamma, \Delta \vdash \textit{expand-aggregate}(e'_1, e'_2) \Rightarrow (e''_1, e''_2), \chi_a, e_a, \chi_\pi, e_\pi \quad \Gamma, \Delta \vdash \textit{expand-product}(e''_1, e''_2) \Rightarrow (\chi_1, \chi_2), e'_\pi}{\begin{array}{c}\Gamma \vdash e_1\ w\ \texttt{ORDER BY}\ e_2\ \texttt{USING}\ e_f : \tau_1 \Rightarrow \\ \Pi_{\chi_1}(\Pi_{e'_\pi/(\chi_1 +\!\!+ \chi_2)}(\mathcal{G}_{e_a/\chi_a}(\Pi_{e_\pi/\chi_\pi}(\mathcal{P}))), \Delta\end{array}}$$

Figure 23: No `GROUP BY` clause

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).

```
1  type 'a t = 'a list
2  val one : 'a t -> 'a
3  val singleton : 'a -> 'a t
4  val product : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
5  val map : ('a -> 'b) -> 'a t -> 'b t
6  val filter : ('a -> bool) -> 'a t -> 'a t
7  val sort : ('a -> 'a -> int) -> 'a t -> 'a t
8  val unique : 'a t -> 'a t
9  val group_all : ('a, 'b) agg -> 'a t -> 'b
10 val group : ('a -> 'c) -> ('a, 'b) agg -> 'a t -> 'b t
11
12 type 'a src = 'a list
13 val input : 'a src -> 'a t
14 val output : 'a t -> 'a src
```

Figure 24: Signature of module `SelectML`

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.

```
1  ⟦ℰ⟧ => SelectML.singleton ()
2
3  ⟦𝒟_χ(e)⟧ => e |> SelectML.input
4
5  ⟦𝒫₁ × 𝒫₂⟧ => SelectML.product
6    (fun ⟦ϕ(𝒫₁)⟧ ⟦ϕ(𝒫₂)⟧ -> ⟦ϕ(𝒫₁)⟧ ++ ⟦ϕ(𝒫₂)⟧) ⟦𝒫₁⟧ ⟦𝒫₂⟧
7
8  ⟦σ_e(𝒫)⟧ => ⟦𝒫⟧ |> SelectML.filter (fun ⟦ϕ(𝒫)⟧ -> e)
9
10 ⟦Π_{e/χ}(𝒫)⟧ => ⟦𝒫⟧ |> SelectML.map (fun ⟦ϕ(𝒫)⟧ -> e)
11
12 ⟦_eS_{e_f}(𝒫)⟧ => ⟦𝒫⟧ |> (let cmp = e_f and key ⟦ϕ(𝒫)⟧ = e in
13     SelectML.sort (fun a b -> cmp (key a) (key b)))
14
15 ⟦𝒰(𝒫)⟧ => ⟦𝒫⟧ |> SelectML.unique
16
17 ⟦_eG_{e₁,...,eₙ}(𝒫)⟧ => ⟦𝒫⟧ |> SelectML.group
18 (fun ⟦ϕ(𝒫)⟧ -> e) ⟦combine(e₁, ..., eₙ)⟧
19
20 ⟦G_{e₁,...,eₙ}(𝒫)⟧ => ⟦𝒫⟧ |> SelectML.group_all ⟦combine(e₁, ..., eₙ)⟧
21     |> SelectML.singleton
22
23 ⟦combine(e₁, ..., eₙ)⟧ =>
24   let Agg (init1, update1, final1) = e₁ in
25   ...
26   let Agg (initn, updaten, finaln) = eₙ in
27   Agg ((init1, ... , initn),
28     (fun (acc1, ... , accn) (x1, ... , xn) ->
29       (update1 acc1 x1, ... , updaten accn xn)),
30     (fun (acc1, ... , accn) -> (final1 acc1, ... , finaln accn)))
31
32 ⟦𝒫 with exactly one row⟧ => ⟦𝒫⟧ |> SelectML.one
33
34 ⟦𝒫 at the outermost⟧ => ⟦𝒫⟧ |> SelectML.output
```

Figure 25: Translation Schema

- For the empty plan $\mathcal{E}$, `SelectML.singleton` is called to construct an idle list with only one row.
- For data source $\mathcal{D}(e)$, `SelectML.input` is used to cast e from type `src` to type `t`.
- For Cartesian product $\mathcal{P} \times \mathcal{P}$, `SelectML.product` is used to produce a product of two lists.
- For selection $\sigma_e(\mathcal{P})$, `SelectML.filter` selects those rows that meet the confition e.
- For projection $\Pi_e(\mathcal{P})$, `SelectML.map` projects the rows from $\phi(\mathcal{P})$ to e.
- For sorting ${}_e\mathcal{S}_{e_f}(\mathcal{P})$, `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 $\mathcal{P}$.
- For deduplication $\mathcal{U}(\mathcal{P})$, `SelectML.unique` is called to deduplicate the rows.
- For grouping ${}_e\mathcal{G}_{e_1,\ldots,e_n}(\mathcal{P})$, `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 $\mathcal{G}_{e_1,\ldots,e_n}(\mathcal{P})$, `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 `('a1, 'b1) agg`, ..., `('an, 'bn) agg`, the result will have type `(('a1, ..., 'an), ('b1, ..., 'bn)) agg`. As data supplied to `SelectML.group` is of type `('a1, ..., 'an) SelectML.t`, 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.one` is called instead of `SelectML.output` 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/parsetree.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 `Pexp_select` and aggregation `Pexp_aggregate` are added as new variants to `Parsetree.expression`.

$$\texttt{Pt.expression} \xrightarrow[\&Planning]{TypeCheck} \texttt{Tt.plan} \xrightarrow{Translating} \texttt{Tt.expression}$$

Figure 26: Typing the Select Expression (`Pt` and `Tt` stands for `Parsetree` and `Typedtree` respectively)

4.1.1 Planning Phase. Originally in the OCaml compiler, the typing phase is to transform `Parsetree.expression` to `Typedtree.expression`. But for `Pexp_select` and `Pexp_aggregate` in *SelectML*, there is a planning phase that will typecheck the given Select Expression and generate a query plan for it.

The query plan is implemented as `Typedtree.plan`, and there is a correspondence to the plans described in Section 3:

- `Tplan_null` for the empty plan $\mathcal{E}$.
- `Tplan_source` for data source $\mathcal{D}_\chi(e)$.
- `Tplan_product` for Cartesian product $\mathcal{P}_1 \times \mathcal{P}_2$.
- `Tplan_filter` for selection $\sigma_e(\mathcal{P})$.

- `Tplan_project` for projection $\Pi_{e/\chi}(\mathcal{P})$.
- `Tplan_sort` for sorting ${}_{e}\mathcal{S}_{e_f}(\mathcal{P})$.
- `Tplan_unique` for deduplication $\mathcal{U}(\mathcal{P})$.
- `Tplan_aggregate_all` for grouping without the `GROUP BY` clause $\mathcal{G}_{e_a/\chi}(\mathcal{P})$.
- `Tplan_aggregate` for grouping ${}_{e}\mathcal{G}_{e_a/\chi}(\mathcal{P})$.

For the aggregation `Pexp_aggregate`, there is a typed version `Texp_aggregate`. However, `Texp_aggregate` is just a temporary form since all the aggregations within the Select Expression are handled during the planning phase. Any standalone occurrence of `Texp_aggregate` outside the Select Expression will be regarded as a compilation error.

4.1.2 Translation and Optimisation. As explained in Section 3.4, query plans are translated into a chunk of function calls to primitives defined in module `SelectML`. Since the compiler searches for the module `SelectML` by name, which means the default primitives can be replaced by shadowing the module with the same name. This leads to a problem that if the searched module lacks the definition of some primitives or the type signatures cannot match, the translated expression will be unsound in type.

```
module type SelectMLType = sig
  type 'a t
  val one : 'a t -> 'a
  val singleton : 'a -> 'a t
  val product : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val sort : ('a -> 'a -> int) -> 'a t -> 'a t
  val unique : 'a t -> 'a t
  val group_all : ('a, 'b) agg -> 'a t -> 'b
  val group : ('a -> 'c) -> ('a, 'b) agg -> 'a t -> 'b t

  type 'a src
  val input : 'a src -> 'a t
  val output : 'a t -> 'a src
end
```

Figure 27: Signature of Primitives

To ensure the translated expressions are well-typed, the module found by the compiler with the name `SelectML` must be checked against the module signature `SelectMLType` (Figure 27) which is defined in file `stdlib/stdlib.ml`.

```
t |> SelectML.input
  |> SelectML.map (fun x -> x)
  |> SelectML.group_all Stdlib.count
  |> SelectML.singleton
  |> SelectML.map (fun __col_2 -> __col_2)
  |> SelectML.one
```

Figure 28: A Translation Result

Figure 28 gives an unoptimised version of the translated code. With the optimisation on removing the unnecessary operations, the plan $\Pi_c(\mathcal{G}_{\text{count}/c}(\Pi_x(\mathcal{D}_x(\text{t}))))$ can be transformed to $\mathcal{G}_{\text{count}/c}(\mathcal{D}_x(\text{t}))$. 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.

```
1  let module SelectML = struct (* Local binding *)
2    include SelectML
3    let group key aggf l =
4      let cmp f a b = compare (f a) (f b) in
5      let Agg (init, update, final) = aggf in
6      l |> 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
10 end in
11 SELECT x FROM x <- [1;2;3;1;2;3] GROUP BY x ORDER BY x;;
12 - : int list = [1; 1; 2; 2; 3; 3]
```

Figure 29: Shadow by Local Bindings

```
SELECT x FROM x <- [1;2;3;1;2;3] GROUP BY x ORDER BY x;;
- : int list = [1; 2; 3]
```

Figure 30: Standard Behaviour

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 constrast, in Figure 31, the output for line 5 and line 8 are the same.

```
1  module SelectML = struct (* Toplevel binding *)
2    include SelectML
3    let group key aggf l = ...
4  end;;
5  SELECT x FROM x <- [1;2;3;1;2;3] GROUP BY x ORDER BY x;;
6  - : int list = [1; 1; 2; 2; 3; 3]
7
8  SELECT x FROM x <- [1;2;3;1;2;3] GROUP BY x ORDER BY x;;
9  - : 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.

```
module SelectML = struct
  include SelectML
  type 'a src = 'a array
  let input = Array.to_list
  let output = Array.of_list
end;;

SELECT x FROM x <- [|1;2;3|] WHERE x mod 2 = 1;;
- : int array = [|1; 3|]

SELECT x, y FROM x <- [|1;2;3|], y <- [|4;5;6|]
ORDER BY x USING odd_first, y USING odd_first;;
- : (int * int) array =
[|(1, 5); (3, 5); (1, 4); (1, 6); (3, 4);
  (3, 6); (2, 5); (2, 4); (2, 6)|]
```

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**.

```
type 'a t = 'a list

let one = function [x] -> x | _ -> assert false
let singleton x = [x]
let product f xs ys =
  List.concat_map (fun x -> List.map (fun y -> f x y) ys) xs

let map = List.map
let filter = List.filter
let sort = List.stable_sort
let unique l = List.sort_uniq compare l

let group_all aggf l =
  let Agg (init, update, final) = aggf in
  final (List.fold_left update init l)

(* Group the rows by sorting them against the group key,
   then aggregate each group. *)
let group key aggf l =
  let cmp f a b = compare (f a) (f b) in
  let Agg (init, update, final) = aggf in
  l |> List.stable_sort (cmp key)
    |> (function
       | [] -> []
       | hd :: tl ->
          let _, l, acc = List.fold_left
            (fun (prev, lst, acc) row ->
              if prev = key row then (prev, lst, update acc row)
              else (key row, final acc :: lst, update init row))
            (key hd, [], update init hd)
            tl
          in final acc :: l)
```

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

```
type 'a t = 'a array

let one t = t.(0)
let singleton x = [|x|]
let product f t1 t2 =
  let n1 = Array.length t1 and n2 = Array.length t2 in
  let len = n1 * n2 in
  Array.init len (fun i -> f t1.(i / n2) t2.(i mod n2))

let map = Array.map
let filter f t =
  let len = ref 0 in
  let b = map (fun x ->
                if f x then (incr len; true) else false) t in
  let j = ref 0 in
  let next _ = while not (b.(!j))
               do incr j done; incr j; t.(!j-1) in
  Array.init !len next

let group_all f t =
  let Agg (init, update, final) = f in
  let acc = ref init in
  for i = 0 to Array.length t - 1
  do acc := update !acc t.(i) done;
  final !acc

(* Group the rows by adding them to a hash table
 * with the group key working as the hash key,
 * and perform the aggregation in the middle of grouping. *)
let group key f t =
  let Agg (init, update, final) = f in
  let ht = Hashtbl.create (Array.length t) in
  for i = 0 to Array.length t - 1 do
    let k = key t.(i) in
    let acc = try Hashtbl.find ht k with Not_found -> init in
    Hashtbl.replace ht k (update acc t.(i))
  done;
  Array.map final (Array.of_seq (Hashtbl.to_seq_values ht))
```

Figure 34: Changing the Intermediate Type

```
module SelectML = struct
  type 'a t = 'a chunk
  let map f t = (* parallel_map_for_chunk ... *)
  let filter f t = (* parallel_filter_for_chunk ... *)
  ...

  type 'a src = 'a table
  let input src = (* cast table to chunk *)
  let output t = (* cast chunk to table *)
end
```

Figure 35: Parallel Implementation

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.

```
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 functor as shown in Figure 37.

```
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 : (* REPL output *)
6    functor (SelectML : SelectMLType) ->
7      sig
8        val run : 'a SelectML.src -> 'a SelectML.src ->
9          ('a * 'a) SelectML.src
10     end
11
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 $[e|(var \leftarrow source)*, condition*]$, 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 [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

[1] [n.d.]. The core OCaml system: compilers, runtime system, base libraries. https://github.com/ocaml/ocaml

[2] [n.d.]. Generalised (SQL-like) List Comprehensions. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/generalised_list_comprehensions.html#generalised-list-comprehensions

[3] [n.d.]. Monad Comprehensions. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/monad_comprehensions.html

[4] [n.d.]. ocaml-caqti. https://github.com/paurkedal/ocaml-caqti

[5] [n.d.]. PostgreSQL Database. https://www.postgresql.org

[6] [n.d.]. SelectML, an SQL frontend on top of OCaml. https://github.com/dyzsr/ocaml-selectml

[7] [n.d.]. SingleStore Database. https://www.singlestore.com

[8] Ludger Becker and Ralf Hartmut Güting. 1992. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems (TODS)* 17, 2 (1992), 247–303. https://doi.org/10.1145/128903.128905

[9] S T Bhosale, Tejaswini Patil, and Pooja Patil. 2015. SQLite: Light Database System. *International Journal of Computer Science and Mobile Computing* 44, 4 (2015), 882–885.

[10] Sebastian Buchwald. 2015. OPTGEN: A generator for local optimizations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9031 (2015), 171–189. https://doi.org/10.1007/978-3-662-46663-6_9

[11] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1773–1786. https://doi.org/10.1145/3299869.3314045

[12] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[13] Marin Fotache and Catalin Strimbei. 2015. SQL and Data Analysis. Some Implications for Data Analysits and Higher Education. *Procedia Economics and Finance* 20, 15 (2015), 243–251. https://doi.org/10.1016/s2212-5671(15)00071-4

[14] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. 2011. Bringing back monad comprehensions. *ACM SIGPLAN Notices* 46, 12 (2011), 13–22. https://doi.org/10.1145/2096148.2034678

[15] Rick Greenwald. 2001. Cost-Based Optimization. *Oracle Internals* (2001), 349–352. https://doi.org/10.1201/9780203997536.ch28

[16] Paolo Guagliardo and Leonid Libkin. 2017. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment* 11, 1 (2017), 27–39. https://doi.org/10.14778/3136610.3136613

[17] Simon Peyton Jones and Philip Wadler. 2007. Comprehensive comprehensions: Comprehensions with 'Order by' and 'Group by'. *Haskell'07: Proceedings of the ACM SIGPLAN 2007 Haskell Workshop* (2007), 61–72. https://doi.org/10.1145/1291201.1291209

[18] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *Proceedings - International Conference on Data Engineering* (2011), 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[19] Xavier Leroy. 2000. A modular module system. *Journal of Functional Programming* 10, 3 (2000), 269–303. https://doi.org/10.1017/S0956796800003683

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010* (2010), 1–10. https://doi.org/10.1109/MSST.2010.5496972

[21] Michael Stonebraker and Greg Kemnitz. [n.d.]. the Postgres Next Generation Dbms. *Language* ([n. d.]).

[22] Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular implicits. *Electronic Proceedings in Theoretical Computer Science, EPTCS* 198 (2015), 22–63. https://doi.org/10.4204/EPTCS.198.2

[23] Wikipedia contributors. 2022. SQL — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SQL&oldid=1082173977 [Online; accessed 21-April-2022].

[24] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

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 $\mathcal{P}\times\mathcal{P}$ together with selection σ 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.t`, it still leads to mediocre performance in most cases.

```
SELECT ... FROM xs JOIN ys ON condition /* inner join */
SELECT ... FROM xs NATURAL JOIN ys      /* natural join */
SELECT ... FROM xs LEFT JOIN ys ON condition /* left outer join */
SELECT ... FROM xs FULL JOIN ys ON condition /* full outer join */
```

Figure 38: Join Operations

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.

```
val join : ('a -> 'b -> 'c option) -> 'a t -> 'b t -> 'c t
val join_eq : ('a -> 'd) -> ('b -> 'd) -> 'a t -> 'b t -> 'c t
```

Figure 39: Primitives for Join Operations

As for future work, a new plan $\mathcal{P}_1 \bowtie_e \mathcal{P}_2$ will be added for the inner join. And new primitives for implementing the inner join (Figure 39).

```
(* when hash key can be determined *)
SELECT ... FROM x <- xs JOIN y <- ys ON x = y;;
  (* translation *)
SelectML.join_eq (fun x -> x) (fun y -> y) xs ys;;

(* when hash key cannot be determined *)
SELECT ... FROM x <- xs JOIN y <- ys ON f x y;;
  (* translation *)
SelectML.join (fun x y->if f x y then Some (x,y) else None) xs ys;;
```

Figure 40: Using Join Operations

When the hash keys of both sides can be determined (Figure 40), plan $\bowtie_e$ 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.

```
/* SQL */
INSERT INTO xs VALUES (1), (2);
INSERT INTO ys VALUES (2), (3);
SELECT x, y FROM xs AS x LEFT JOIN ys AS y ON x = y;

 x | y
---+---
 1 |
 2 | 2
```

```
(* SelectML *)
let xs = [1; 2];;
let ys = [2; 3];;
SELECT x, y FROM x <- xs LEFT JOIN y <- ys ON x = y;;

 x |   y
---+--------
 1 | None
 2 | Some 2
```

Figure 41: Example of Left Outer Join

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 perserve the data from the left side, right outer joins perserve the data from the right side, and full outer joins perserve both sides. Empty cells will be filled with **NULL**s. 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.

```
SELECT x, y FROM x <- xs RIGHT JOIN y <- ys ON x = y;;

    x    | y
---------+---
 Some 2  | 2
 None    | 3

SELECT x, y FROM x <- xs FULL JOIN y <- ys ON x = y;;

    x    |   y
---------+--------
 Some 2  | Some 2
 Some 1  | None
 None    | Some 3
```

Figure 42: Example of Right/Full Outer Join

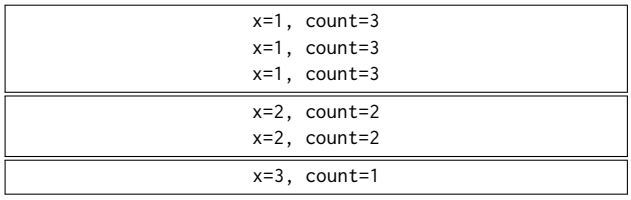
A.2 Window Functions

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. An example of window functions in PostgreSQL 14.2 is shown in Figure 43. 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**.

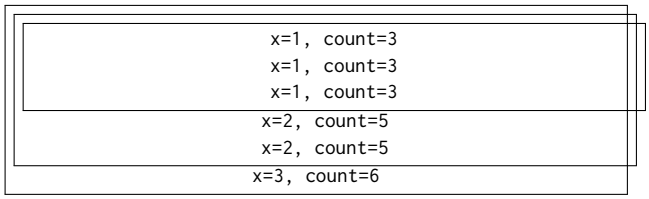
```
INSERT INTO t VALUES (1,1),(1,2),(1,3),(2,2),(2,3),(3,3);
SELECT x, COUNT(y) OVER (PARTITION BY x) FROM t;
SELECT x, COUNT(y) OVER (ORDER BY x) FROM t;

/* PARTITION BY */    /* ORDER BY */
 x | count              x | count
---+-------            ---+-------
 1 |     3              1 |     3
 1 |     3              1 |     3
 1 |     3              1 |     3
 2 |     2              2 |     5
 2 |     2              2 |     5
 3 |     1              3 |     6
```

Figure 43: Example of Window Functions



(a) Window Frames of PARTITION BY



(b) Window Frames of ORDER BY

Figure 44: Example of Windows Frames

Figure 44 gives an overview of the window frames generated by different window specifications for Figure 43. In *SelectML*, the syntax for window function can be tailored to *aggregation* **OVER** *window-spec*, like in Figure 45.

```
{agg y} OVER (PARTITION BY x)
{agg y} OVER (ORDER BY x)
{agg y} OVER (ROWS UNBOUNDED PRECEDING)
```

Figure 45: Window Functions in *SelectML*

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.

```
type ('a, 'b, 'c) aggfunc =
  'c *                (* the initial value *)
  ('c -> 'a -> 'c) *  (* append to the tail *)
  ('a -> 'c -> 'c) *  (* remove from the head *)
  ('c -> 'b)          (* get the result *)

type (_, _) agg = Agg : ('a, 'b, 'c) aggfunc -> ('a, 'b) agg
```

Figure 46: Aggregate/Window Functions

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: $\Pi_{(c_1,c_2)}(\Pi_{(x,y)/(c_1,c_2)}(\mathcal{P})) \Longrightarrow \Pi_{(x,y)}(\mathcal{P})$.
- pushing down the selection in order to save the computation on joining:

$$\sigma_{x<1 \wedge y>1}(\mathcal{D}_x(\text{xs}) \times \mathcal{D}_y(\text{ys})) \Longrightarrow \sigma_{x<1}(\mathcal{D}_x(\text{xs})) \times \sigma_{y>1}(\mathcal{D}_y(\text{ys}))$$

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.

```
1 /* SQL */
2 SELECT x FROM xs AS x WHERE x BETWEEN lb AND ub
3 SELECT x FROM xs AS x, ys AS y WHERE x < 1
4 SELECT x FROM xs AS x, ys AS y WHERE x < y
```

Figure 47: Condition Involving Comparisons

Indexes is an important optimisation point when there are comparisons in the `WHERE` condition (Figure 47). Without indexes on `x`, plan $\sigma_{x<1}(\mathcal{D}_x(\text{xs}))$ 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 $(-\inf, 1)$. 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 `(int * int) 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 *datatype* `ORDER BY` *fields*. For tuple types like `(int * int) list` (line 16, 17), the *fields* are integers to denote which columns are to be indexed , i.e. `0` denotes the first column of `(int * int) list`. For records like `order list` (line 21, 22), the *fields* should be the field names of the record type.

```
1  (* SelectML *)
2  type index
3
4  module type SelectMLType = sig
5    ...
6    val get : index -> 'a t -> 'a
7  end;;
8
9  module type IndexType = sig
10   type 'a t
11   val data : 'a SelectML.t
12   val to_seq : 'a t -> (index * 'a) Seq.t
13   val to_seq_from : 'a -> 'a t -> (index * 'a) Seq.t
14 end;;
15
16 let f (module XS : (int * int) list ORDER BY 0) =
17   SELECT x, y FROM (x, y) <- (module XS) WHERE lb <= x && x <= ub;;
18
19 type order = { id: int; price: float; month: string }
20
21 let f' (module XS : order list ORDER BY price) =
22   SELECT x FROM x <- (module XS) WHERE lb <=. x && x <=. ub;;
```

Figure 48: Typing Indexes

The `FROM` clause is responsible for recognising the index type in the source expression. Syntax *datatype* `ORDER BY` *fields* serves as module interfaces, any module passed as arguments must be an instance of `IndexType`. `type 'a t` stands for the data structure storing the indexes (line 10). `data` refers to the original data of type `SelectML.t` (line 11). `to_seq` and `to_seq_from` are analogous to functions defined in OCaml `Map`, which output a sequence of ordered indexes (line 12, 13). With the indexes, data can be retrieved using `SelectML.get` (line 6).

For instance, `lb <= x` (line 17) will be translated to `XS.to_seq lb XS.data` to get the indexes sequence starting from `lb`. The retrieving of data will stop when `ub` is met.