An SQL Frontend on top of OCaml for Data Analysis

Yan Dong, Yahui Song, Chin Wei Ngan

National University of Singapore

2nd Sep 2022 @ IFL'22 in Copenhagen

From loops to functional stream processing

```
1       List<Album> albums = ...;
2       List<String> result = new ArrayList<>();
3
4       for (Album album : albums) {
5
6           if (album.getYear() != 1999) {
7               continue;
8           }
9
10          if (album.getGenre() != Genre.ALTERNATIVE) {
11              continue;
12          }
13
14          result.add(album.getName());
15
16          if (result.size() == 5) {
17              break
18          }
19      }
20
21      Collections.sort(result);
```

This code is equivalent to:

```
1       List<String> result =
2           albums.stream()
3                 .filter(album -> album.getYear == 1999)
4                 .filter(album -> album.getGenre() == Genre.ALTERNATIVE)
5                 .limit(5)
6                 .map(Album::getName)
7                 .sorted()
8                 .collect(Collectors.toList());
```

Motivation Example - Data Processing

- Incomes sorted by month in decreasing order in 2021

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

OCaml code written in a functional manner

With SQL Select Query, the logic can be rewritten in a more concise way

```
1  let income_by_month : order list -> (string * float) list =
2    fun orders ->
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
```

Motivation Example - Data Processing

- Incomes sorted by month in decreasing order in 2021

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

Implement databases by modifying general-purposed programming languages

OCaml code written in a functional manner

With SQL Select Query, the logic can be rewritten in a more concise way

```
1  let income_by_month : order list -> (string * float) list =
2    fun orders ->
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
```

We propose: *SelectML = OCaml + Select Query*

- **SelectML is built on top of OCaml**
  - Static typed & Functional
- **SelectML introduces SQL Select Query onto OCaml**
  - Supporting a declarative way for data processing
- **Suitable for operating list data**
  - Also supporting arrays, sequences, and user-defined data types
- **Orthogonal with other OCaml language features**
  - Core language, module language, object language, …

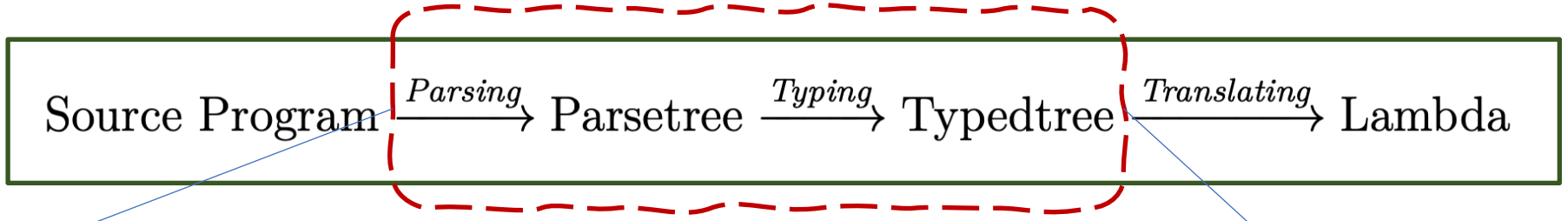
System Overview

- **OCaml frontend**

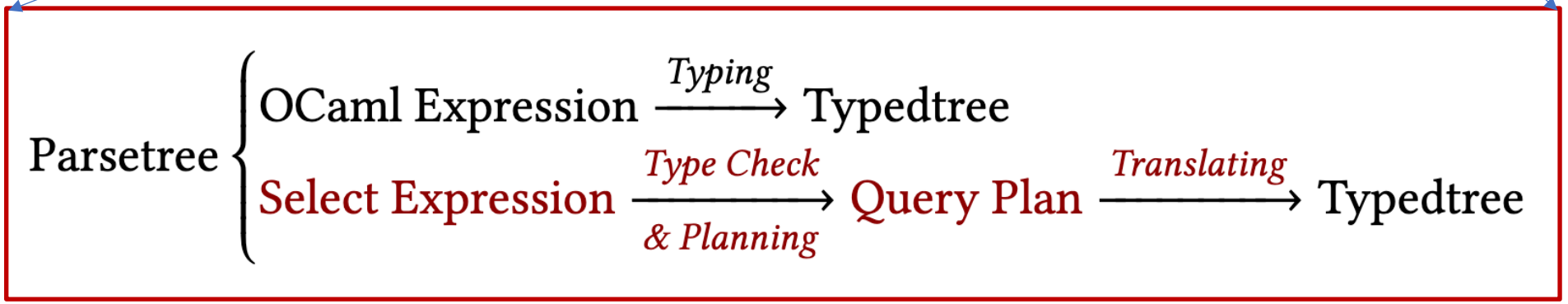
- **SelectML is implemented by modifying the parsing and typing phases**

System Overview

- **OCaml frontend**



- **SelectML is implemented by modifying the parsing and typing phases**



✓Adding the syntax support for SelectML

✓Conduct type checking before code refracturing

✓Code refracturing via a formally defined translation schema with a set of query plans

SelectML Language Design

- **OCaml + Select Expression**
  - SELECT, DISTINCT, FROM, WHERE, GROUP BY, HAVING, ORDER BY
  - Aggregate functions & applications
- **Common features for data analysis**
  - mapping, filtering, grouping, ordering

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

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

- **FROM clause**
  - `xs AS x` changed to ‘`x←xs`’
  - `x::'a, xs::'a list`

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

```
(* SelectML *)
SELECT x, y FROM x <- xs, y <- ys;;
```

- **GROUP BY clause**
  - `COUNT(y)` changed to `{count y}`

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

```
(* SelectML *)
SELECT x, {count y} FROM (x, y) <- t GROUP BY x;;
```

- **FROM clause**
  - `xs AS x` changed to '`x←xs`'
  - `x::'a, xs::'a list`

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

```
(* SelectML *)
SELECT x, y FROM x <- xs, y <- ys;;
```

- **GROUP BY clause**
  - `COUNT(y)` changed to `{count y}`

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

```
(* SelectML *)
SELECT x, {count y} FROM (x, y) <- t GROUP BY x;;
```

- **WHERE & HAVING clause**
  - Boolean expressions
  - WHERE: before GROUP BY
  - HAVING: after GROUP BY

```
SELECT x FROM x <- xs WHERE f x;;
SELECT {h x} FROM x <- xs HAVING g {h x};;
```

- **Aggregate functions**
  - Line 2:
    `('a, 'b) agg` is the built-in type for aggregate functions
  - Line 4~9:
    Common aggregate functions: AVG, COUNT, SUM, MIN, MAX

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

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];;
```

- **ORDER BY clause**
  - Ordering key must be comparable expressions
  - Ordering directions: `ASC`, `DESC` , `USING cmp_func`

```
let odd_first a b =
    let x = a mod 2 = 0 in
    let y = b mod 2 = 0 in
    compare x y;;

SELECT x, y
FROM (x, y) <- [("a", 2); ("a", 3); ("b", 4); ("b", 5)]
ORDER BY x ASC, y USING odd_first;;

-:(string * int) SelectML.src=[("a",3); ("a",2);("b",5);("b",4)]
```

- **ORDER BY clause**
  - Ordering key must be comparable expressions
  - Ordering directions: `ASC`, `DESC` , `USING cmp_func`

```
let odd_first a b =
    let x = a mod 2 = 0 in
    let y = b mod 2 = 0 in
    compare x y;;

SELECT x, y
FROM (x, y) <- [("a", 2); ("a", 3); ("b", 4); ("b", 5)]
ORDER BY x ASC, y USING odd_first;;

-:(string * int) SelectML.src=[("a",3); ("a",2);("b",5);("b",4)]
```

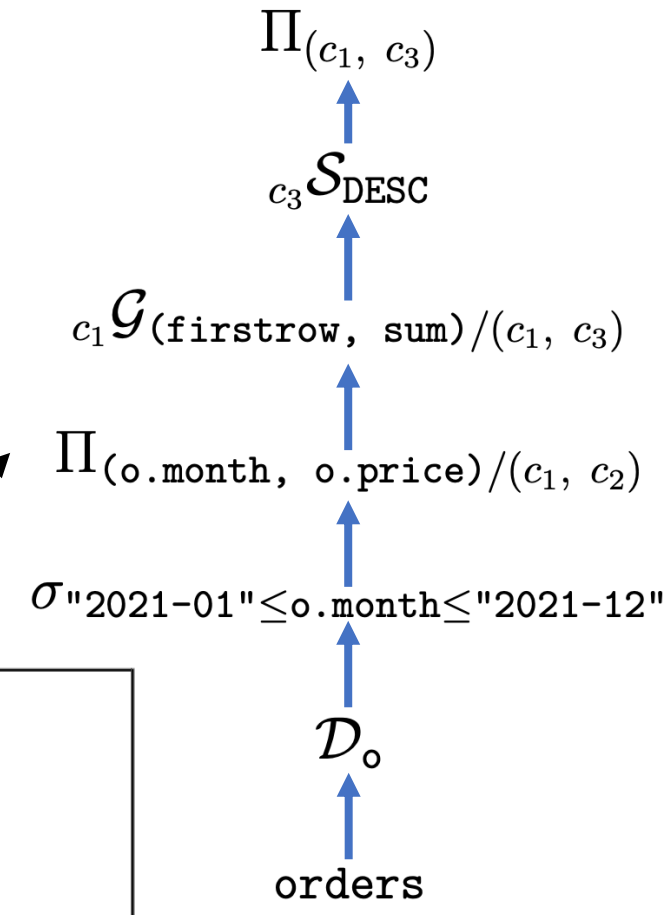
- **SELECT clause**
  - DISTINCT for deduplication
- **Type of Select Expressions**
  - Assume the type of SELECT clause is `'a`
  - Line 1~8: general cases
    - type `'a list`
  - Line 10~16: returns exactly one row
    - type `'a`

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

SELECT DISTINCT x FROM x <- [1;1;2;2;3;3];;
- : int list = [1; 2; 3]

SELECT y, x FROM (x, y) <- [("a", 1); ("b", 2)];;
- : (int * string) list = [(1, "a"); (2, "b")]

(* without FROM, WHERE, and HAVING *)
SELECT x, y;;
SELECT x, y GROUP BY z;;
SELECT x, y ORDER BY z;;

(* aggregation without GROUP BY, WHERE, and HAVING *)
SELECT {count x} FROM x <- t;;
```

Query Plans and Translation Schema

- **Source code → Query plans (8 kinds)**

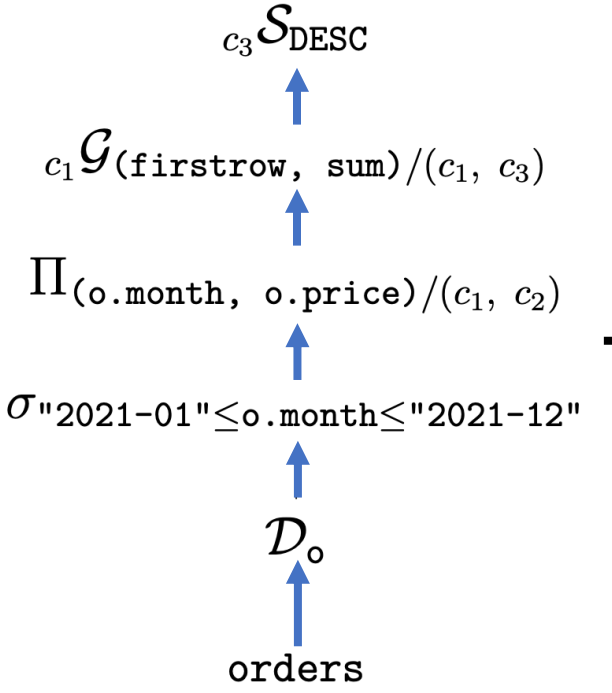
$\mathcal{E}$	Empty Source	$\sigma_e(\mathcal{P})$	Selection
$\mathcal{D}_\chi(e)$	Data Source	$\Pi_{e/\chi}(\mathcal{P})$	Projection
$\mathcal{P}_1 \times \mathcal{P}_2$	Cartesian Product	${}_e\mathcal{G}_{e_a/\chi}(\mathcal{P})$	Aggregation
$\mathcal{U}(\mathcal{P})$	Deduplication	${}_e\mathcal{S}_{e_f}(\mathcal{P})$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




```
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
```

Query Plans and **Translation Schema**

- **Translation Schema: Query plans → Typed OCaml code**
  - Semantics of the query plans are captured by primitives listed in module `SelectML`




```
1  orders
2  |> SelectML.input
3  |> SelectML.filter (fun o ->
4                       o.month >= "2021-01" && o.month <= "2021-12")
5  |> SelectML.map (fun o -> o.month, o.price)
6  |> SelectML.group (fun (__col_1, __col_2) -> __col_1)
7       (let Agg (init1, update1, final1) = Stdlib.firstrow in
8        let Agg (init2, update2, final2) = Stdlib.sum in
9        Agg ((init1, init2),
10         (fun (acc1, acc2) (x1, x2) ->
11             update1 acc1 x1, update2 acc2 x2),
12         (fun (acc1, acc2) -> final1 acc1, final2 acc2)))
13 |> let key (__col_1, __col_3) -> __col_3) in
14    SelectML.sort (fun a b -> compare (key b) (key a))
15 |> SelectML.output
```

Translation Schema in Detail

- Query plans are translated to primitives defined in Stdlib module `SelectML`
- Primitives defined on line 1~10 are used by intermediate computations
- Primitives defined on line 12~14 are used for casting type

```
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
```

Translation Schema in Deta

- Query plans are translated to primitives defined in St
- Primitives defined on line 1~10 are used by intermedi
- Primitives defined on line 12~14 are used for casting

```
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 ->
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 ->
11
12 type 'a src = 'a list
13 val input : 'a src -> 'a t
14 val output : 'a t -> 'a src
```

```
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
```

Implementation

- **Core implementation:** ~ 1000 LOC in OCaml
- **Test for validation:** ~ 60 testcases, manually marked with the expected outputs
- **Flexible Customisations**
  - ✓ Customise the semantics of Select Expression: rewrite the WHERE implementation to keep the falsas
  - ✓ Change the input and output type of Select Expression: from `'a list` to `'a array`
  - ✓ Change the intermediate type of Select Expression: from list to array.
  - ✓ Generalization with Functors: to deal with both list type and array type.

https://github.com/dyzsr/ocaml-selectml

Implementation

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

This does not type check!!!

Figure 36: Generalising the Type and Operations

- **Core implementation:** ~ 1000 LOC in OCaml
- **Test for validation:** ~ 60 testcases, manually marked with the expected outputs
- **Flexible Customisations**
  - ✓Customise the semantics of Select
  - ✓Change the input and output type
  - ✓Change the intermediate type of S
  - ✓Generalization with Functors: to de

https://github.com/dyzsr/o

```
module F (SelectML : SelectMLType) = struct
  let run xs ys = SELECT x, y FROM x <- xs, y <- ys WHERE x < y;;
end;;

module F : (* REPL output *)
  functor (SelectML : SelectMLType) ->
    sig
      val run : 'a SelectML.src -> 'a SelectML.src ->
        ('a * 'a) SelectML.src
    end

(* Supplying primitives with the list implementation *)
let open F (ListImpl) in run ...

(* Supplying primitives with the array implementation *)
let open F (ArrayImpl) in run ...
```

Possible More Features

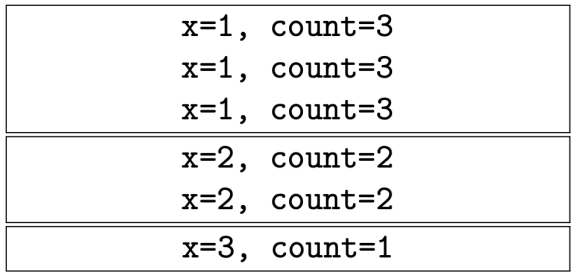
1. Joins ⇔ Cartesian product + filter: may reduce memory consumption

```
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
```

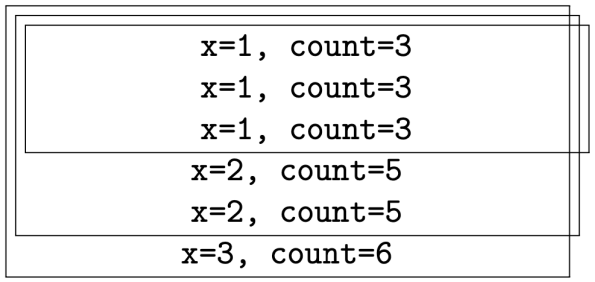
```
(* when hash key can be determined *)
SELECT ... FROM x <- xs JOIN y <- ys ON x = y;;
  (* translation *)
SelectML.join_eq (fun x -> x) (fun y -> y) xs ys;;

(* 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;;
```

2. Window Functions

- They are like aggregate functions, but without causing rows to become grouped into a single output
- Additional window functions: LEAD, LAG, RANK, ...



(a) Window Frames of `PARTITION BY`



(b) Window Frames of `ORDER BY`

```
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
```

Possible Optimizations

1. Rule-based optimisation

- Eliminate unnecessary plans $\Pi_{(c_1,c_2)}(\Pi_{(x,y)/(c_1,c_2)}(\mathcal{P})) \Longrightarrow \Pi_{(x,y)}(\mathcal{P})$
- Push down filters $\sigma_{x<1 \wedge y>1}(\mathcal{D}_x(\mathrm{xs}) \times \mathcal{D}_y(\mathrm{ys})) \Longrightarrow \sigma_{x<1}(\mathcal{D}_x(\mathrm{xs})) \times \sigma_{y>1}(\mathcal{D}_y(\mathrm{ys}))$

2. Cost-based optimisation

- Select the optimal algorithm for a query plan, based on the runtime information of data

3. Indexes

- Optimising table scan
- May require modifications

  on the OCaml type system

```
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
```

```
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;;
```

Create Database

- **From REPL** (Read–eval–print loop) **to Database**
  - Behaviors
    - #CREATE introduces a top-level variable `xs` of type `(int * int) table`
    - #INSERT inserts two new rows `(1,2), (3,4)` to `xs`
    - #UPDATE updates `xs` by setting increasing the first column by 1

```
#CREATE TABLE xs : int * int;;
#INSERT INTO xs VALUES (1,2), (3,4);;
#UPDATE (a, b) <- xs SET (a+1, b);;

SELECT a, b FROM (a, b) <- xs WHERE a < 4;;
```

```
SelectML.insert [(1,2); (3,4)] xs;;
SelectML.update (fun (a, b) -> (a+1, b)) xs;;
```

```
module SelectMLType = sig
  ...
  val insert : 'a list -> 'a src -> unit
  val update : ('a -> 'a) -> 'a src -> unit
end
```

Conclusions

- SelectML = OCaml + Select Query
- Language Design: Select Expression & Aggregate functions
- Semantics: Typing & Planning rules & Translation schema
- Implementation: Different Flexible Customisations
- More possible features and future work
  - Joins, window functions, optimisations, indexes
  - Connection to database via libraries such as 'caqti' with 'postgresql' database

Conclusions

- SelectML = OCaml + Select Query
- Language Design: Select Expression & Aggregate functions
- Semantics: Typing & Planning rules & Translation schema
- Implementation: Different Flexible Customisations
- More possible features and future work
  - Joins, window functions, optimisations, indexes
  - Connection to database via libraries such as 'caqti' with 'postgresql' database

Thanks!