Optimizing SQL UDFs with MLIR

CARNEGIE MELLON DATABASE GROUP

Carnegie Mellon University
School of Computer Science

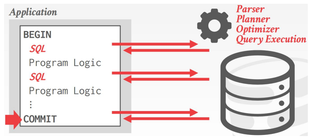
Rohan Aggarwal (rohanagg@andrew.cmu.edu), Amadou Ngom (angom@andrew.cmu.edu)

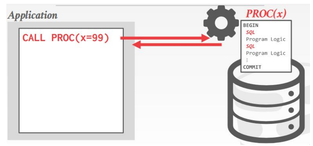
Introduction and Background

- DBMSs provide a way to implement server-side logic by writing UDFs.
  - Mixed imperative and relational.

```
FUNCTION EuroToDollar(money):
  DECLARE rate = SELECT rate from rates
                 WHERE origin='euro'
                 AND dest='dollar';
  RETURN rate * money;
```

  - Reduced Overhead.





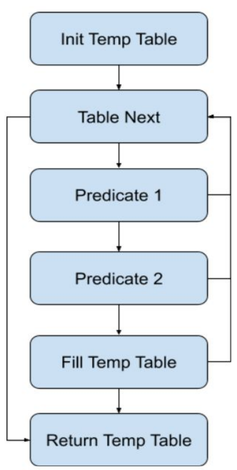
- Traditional UDF execution context switches between imperative and relational execution.
  - Previous work found that context switching causes significant overhead.
- Recent solutions: Imperative to relational conversion, optimize, then execute.
  - Currently limited to scalar-valued functions.
- Our approach: Convert UDF custom high level IR (SQLIR), optimize, then execute.

SQLIR

```
func @Select(%arg0: i64) -> i64 attributes {IsSelect = True} {
  %0 = sqlir.NewTempTable  : () -> i64
  br ^bb1
^bb1: // 4 preds: ^bb0, ^bb2, ^bb3, ^bb4
  %c37_i64 = constant 37 : i64
  %1 = sqlir.TableNext %c37_i64 : (i64) -> i1
  cond_br %1, ^bb2, ^bb5
^bb2: // pred: ^bb1
  %c37_i64_0 = constant 37 : i64
  %c1_i64 = constant 1 : i64
  %2 = sqlir.GetColumn %c37_i64_0, %c1_i64 : (i64, i64) -> i64
  %c37_i64_1 = constant 37 : i64
  %c2_i64 = constant 2 : i64
  %3 = sqlir.GetColumn %c37_i64_1, %c2_i64 : (i64, i64) -> i64
  %4 = cmpi "slt", %2, %3 : i64
  cond_br %4, ^bb3, ^bb1
^bb3: // pred: ^bb2
  %c37_i64_2 = constant 37 : i64
  %c73_i64 = constant 73 : i64
  %5 = cmpi "sgt", %c37_i64_2, %c73_i64 : i64
  cond_br %5, ^bb4, ^bb1
^bb4: // pred: ^bb3
  %c37_i64_3 = constant 37 : i64
  %c37_i64_4 = constant 37 : i64
  %6 = muli %c37_i64_3, %c37_i64_4 : i64
  sqlir.FillResult %6, %0 : (i64, i64) -> ()
  %c37_i64_5 = constant 37 : i64
  %c1_i64_6 = constant 1 : i64
  %7 = sqlir.GetColumn %c37_i64_5, %c1_i64_6 : (i64, i64) -> i64
  %c37_i64_7 = constant 37 : i64
  %c2_i64_8 = constant 2 : i64
  %8 = sqlir.GetColumn %c37_i64_7, %c2_i64_8 : (i64, i64) -> i64
  %9 = muli %7, %8 : i64
  sqlir.FillResult %9, %0 : (i64, i64) -> ()
  br ^bb1
^bb5: // pred: ^bb1
  return %0 : i64
}
```

```
func @Main(%arg0: i64) -> i64 attributes {IsSelect = false} {
  %cst = constant 3.777000e+01 : f64
  %c37_i64 = constant 37 : i64
  %0 = call @Select(%c37_i64) : (i64) -> i64
  %1 = call @Join() : () -> i64
  %c0_i64 = constant 0 : i64
  %c0_i64_0 = constant 0 : i64
  %2 = sqlir.FetchValue %0, %c0_i64, %c0_i64_0 : (i64, i64, i64) -> i64
  %c0_i64_1 = constant 0 : i64
  %c0_i64_2 = constant 0 : i64
  %3 = sqlir.FetchValue %1, %c0_i64_1, %c0_i64_2 : (i64, i64, i64) -> i64
  %4 = muli %2, %3 : i64
  return %4 : i64
}
```

```
SELECT 37*37, col1*col2
FROM FOO WHERE
col1 < col2
AND
37 < 73
```



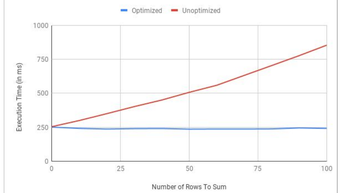
Imperative Optimizations on Selects

- Using MLIR, we get classical imperative optimizations for free.
- Select statements possess statistics on tables.
  - We can perform traditionally unsafe optimization.

LICM

```
var result = SELECT expansive_call(const_arg1) * col1
             FROM table
             WHERE col2 < val;
```

```
var precomputed = expansive_call(const_arg1);
var result = SELECT precomputed * col1
             FROM table
             WHERE col2 < val;
```

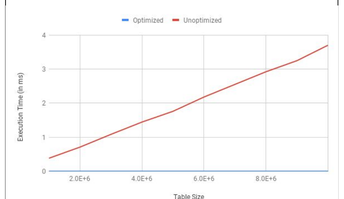


DCE

```
var result = SELECT *
             FROM large_table
             WHERE false_predicate;
```

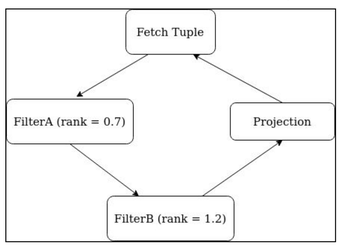
```
func @Select(%arg0: i64) -> i64 attributes {IsSelect = True} {
  %0 = sqlir.NewTempTable  : () -> i64
  return %0 : i64
}
```

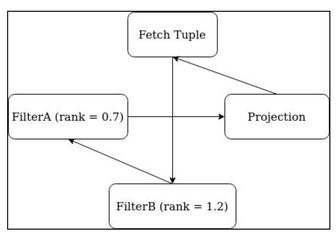
```
var result = EMPTY;
```



Filter Ordering

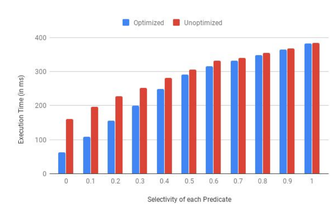
- Given the structure of our IR, we can easily order filters by changing branch targets.
  - Ordered by rank: (1 - selectivity) / cost-per-tuple.
  - To avoid statistics collection, we only implement 1 / cost-per-tuple.

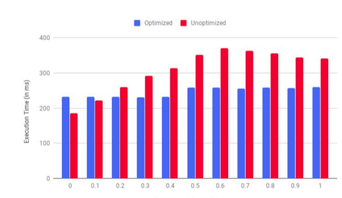





```
var result = SELECT *
             FROM table
             WHERE (col1 % col2 > val)
             AND   (col3 > val);
```

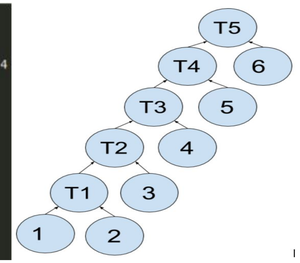
```
var result = SELECT col1*col2
             FROM FOO WHERE
             col1 < val1          #fixed
             AND
             col2 < val2          #varying
```



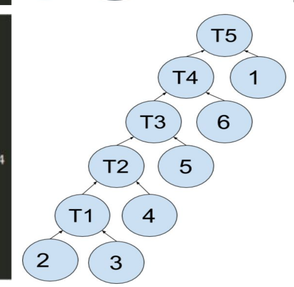


Join Ordering

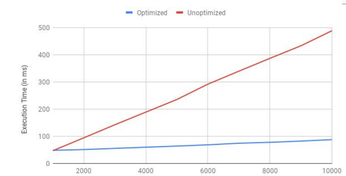
```
func @Join() -> i64 attributes {IsSelect = false} {
  %c1_i64 = constant 1 : i64
  %c2_i64 = constant 2 : i64
  %0 = sqlir.Join %c1_i64, %c2_i64 : (i64, i64) -> i64
  %c3_i64 = constant 3 : i64
  %1 = sqlir.Join %c3_i64, %0 : (i64, i64) -> i64
  %c4_i64 = constant 4 : i64
  %2 = sqlir.Join %c4_i64, %1 : (i64, i64) -> i64
  %c5_i64 = constant 5 : i64
  %3 = sqlir.Join %c5_i64, %2 : (i64, i64) -> i64
  %c6_i64 = constant 6 : i64
  %4 = sqlir.Join %c6_i64, %3 : (i64, i64) -> i64
  %c7_i64 = constant 7 : i64
  %5 = sqlir.Join %c7_i64, %4 : (i64, i64) -> i64
  return %5 : i64
}
```



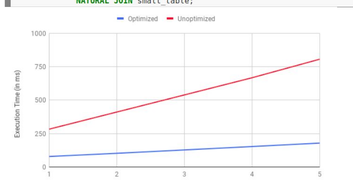

```
func @Join() -> i64 attributes {IsSelect = false} {
  %c1_i64 = constant 1 : i64
  %c2_i64 = constant 2 : i64
  %c3_i64 = constant 3 : i64
  %c4_i64 = constant 4 : i64
  %c5_i64 = constant 5 : i64
  %c6_i64 = constant 6 : i64
  %c7_i64 = constant 7 : i64
  %0 = sqlir.Join %c2_i64, %c3_i64 : (i64, i64) -> i64
  %1 = sqlir.Join %c7_i64, %0 : (i64, i64) -> i64
  %2 = sqlir.Join %c4_i64, %1 : (i64, i64) -> i64
  %3 = sqlir.Join %c5_i64, %2 : (i64, i64) -> i64
  %4 = sqlir.Join %c6_i64, %3 : (i64, i64) -> i64
  %5 = sqlir.Join %c1_i64, %4 : (i64, i64) -> i64
  return %5 : i64
}
```




```
var result = SELECT *
             FROM large_table
             NATURAL JOIN medium_table
```




```
var result = SELECT *
             FROM large_table
             NATURAL JOIN medium_table
             NATURAL JOIN small_table;
```



Future Work and Conclusion

- We have not implemented all relational operators, and have not tried to efficiently compose them.
- In addition, this approach could be used for other suchs as Dataframes.
- Optimization framework performs optimizations in a general way combining the optimizations found both in the imperative and the declarative world.
- UDFs lie at the intersection of these two worlds and it's important that we are not restricted while optimizing them.
- Able to leverage optimizations that only appear when both these worlds combine like DCE for Select Statements.
- Did not lose any optimizations in the Declarative world - Join Ordering, Filter Ordering.
- Lot of potential for future work.
- Happy that built this from ground up…!!!