Unified UDF Compilation

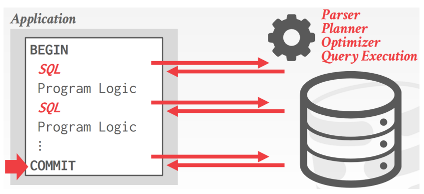
Rohan Aggarwal and Amadou Ngom

Carnegie Mellon University

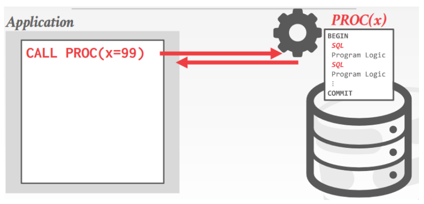
Abstract. To avoid the overhead network round trips, Database Management Systems (DBMSs) provide the functionality of user-defined, server-side functions (UDFs). These functions mix imperative constructs with relational ones (e.g., scans, joins) to provide maximal flexibility to programmers. However, UDFs have historically been notoriously slow for two reasons due cost of context switching between relational code and imperative one. Recent work has focused on the following steps: convert imperative code into purely relational code through fairly sophisticated techniques, optimize the relational code, then compile the relational code back into imperative code to execute it. These approaches are currently limited to scalar-valued UDFs, i.e., functions that return a single value rather than a table. In this experimental project, we try out a different strategy: express the UDF in a simple, high-level intermediate representation (IR) using MLIR, directly optimize this IR, lower it to executable machine code. We show that we can perform both relational and imperative optimizations on scalar and table-valued UDFs.

Keywords: UDF, MLIR, Optimizations

1 Introduction and Background



(a) Client Side Logic [3]



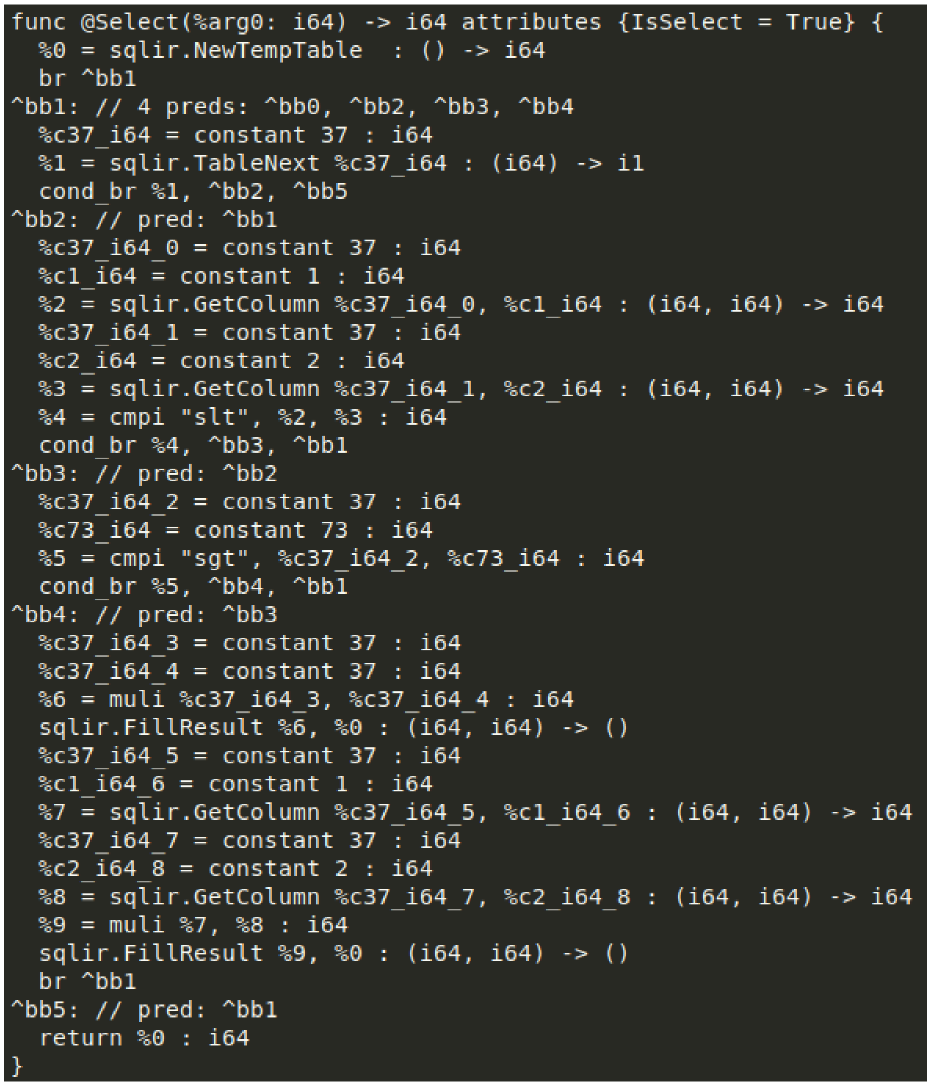
(b) Server Side Logic [3]

Fig. 1: Client versus Server side logic.

Client-side logic frequently needs to transfer SQL queries to the server to execute them and get back a result. This network round trip overhead may compound with the need to parse, optimize, compile, and execute queries that don't happen to be cached (Figure 1a). User-Defined Functions (UDF) address this issue by implementing the logic directly on the server using languages that mix imperative (e.g., assignments, branches, loops) and relational (e.g., scans, joins, aggregations) constructs (Figure 1b). Traditional query engines execute UDFs by using an imperative interpreter for imperative code, and by performing a context switch whenever they encounter a declarative construct. The cost of switching between these two execution modes is substantial [4]. Recent work has tried to fully convert imperative constructs into relational ones before optimizing and compiling the whole UDFs into one executable module [4,7]. This approach is currently limited to scalar-valued UDFs (SVFs). We experiment with a different approach that works with both SVFs and table-valued UDFs (TVFs). We first convert the UDFs into our custom, high-level IR called SQLIR that's a dialect of MLIR [6]. MLIR is a compiler framework that facilitates the development of domain-specific IRs. A 'dialect' of MLIR is simply an abstraction to define the constructs of our domain (e.g., scanning a table, joining them). We then perform both relational optimizations such as filter ordering and join ordering and imperative optimizations such as dead code elimination (DCE) and loop invariant code motion (LICM) directly on this IR.

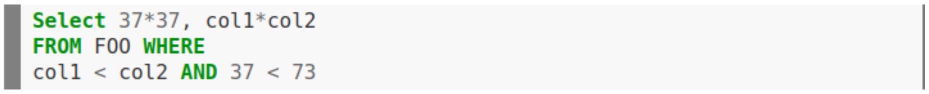
2 SQLIR

SQLIR is an attempt to bridge the gap between relational and imperative optimizations and have a single optimizer in the database framework that can work well with general UDFs and the queries inside them. SQLIR is an intermediate DSL that has been constructed taking into account the wide array of optimizations that can be leveraged while optimizing a UDF.
Each SQL construct is referred through a function call that mimics the substructure corresponding to the declarative clause in SQL. Every SQLIR code has a main function that represents UDF being processed. Every Declarative query inside this refers to a function present in the file and we call the function with necessary arguments whenever required. The functions can be called recursively too.
First, we would like to show the SQLIR for a very basic SELECT query.



```
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
}
```

Fig. 2: Basic Select SQLIR Function



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

Fig. 3: Basic Select Query

Figure 2 shows the SQLIR function corresponding to the Select query in Figure 3. We would like to highlight some salient points here to explain our structure.

1. The input argument to the Select function is the table_oid for which we are performing the select on. This is replaced when the binder looks for table FOO in the catalog and finds its table_oid. In this case it is 37.

2. The return argument represents the table_oid of the temp table where we materialized the results. This temporary space is created by calling the function NewTempTable at the beginning of the function.

3. For select statement involving multiple tables in the FROM clause, the input argument will be the table_id of a temp table which is outputted by a join function. More on this later.

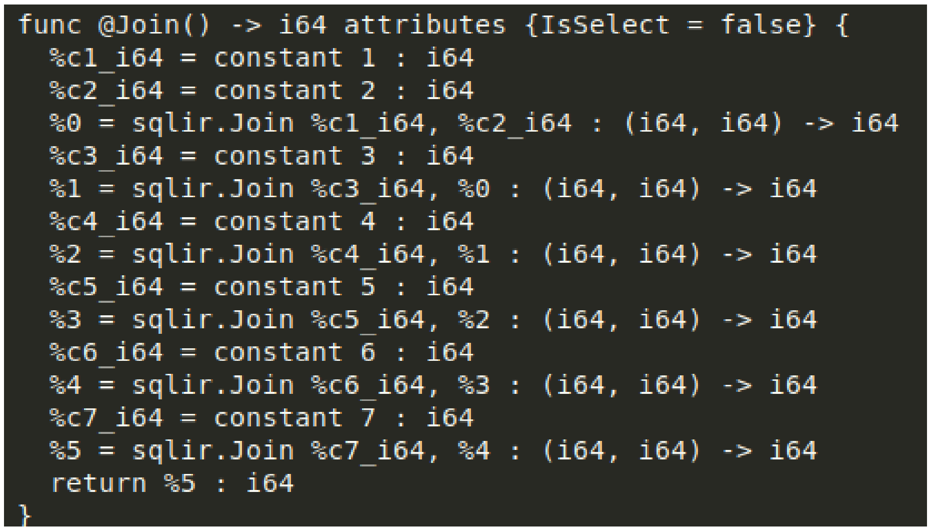
4. The basic block bb1 represents the part of the statement where we go through each tuple of the table. Whenever TableNext returns false we stop reading and jump to the exit block bb5 and return the temp table identifier.

5. bb2 and bb3 represent the two predicates in our select statement. bb3 is a constant comparison which is always true and hence we do not need much here. For bb2 we use the GetColumn function with column_oid of the columns(again determined by the binder) which returns the attribute of the current tuple with the given column_oid. The GetColumn Function takes two arguments the table_oid and the column_oid.

6. bb4 represents the part where we full the results in the temp table. Since we have already crossed all predicates we can safely insert in our temp table. For this we use the FillResult function, which takes in two arguments the attribute to be filled and the temp table identifier.

7. The IsSelect attribute is used while doing optimization passes to determine the type of function.

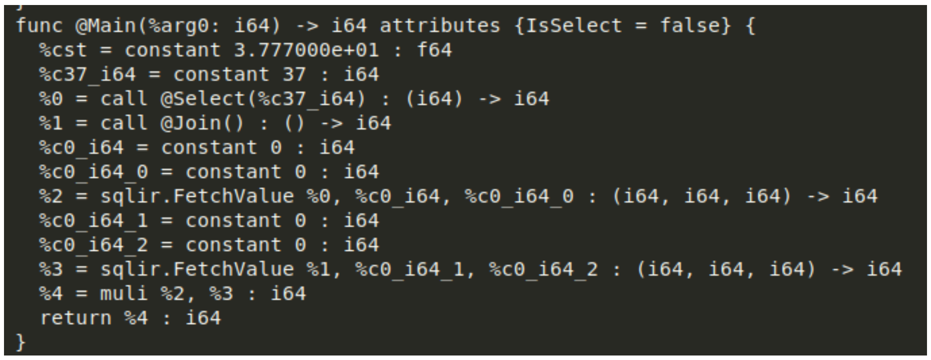
Now we will talk about the Join Function which represents the join of multiple tables in sqlir.



```
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
}
```

Fig. 4: Basic Join Operation

The Join Operation in Figure 4 is relatively straightforward. It represents the join of multiple tables which are referred by their table identifiers determined by the binder. It returns a single argument which represents the temp table that contains the result of the join. Right now our join function only supports Natural joins, but it is easy to see that predicate evaluation is not that tough to support given the framework that we have already set up. Note that the return argument of the Join Operation can be fed as an input argument to a select query that has multiple tables in the FROM clause.



```
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
}
```

Fig. 5: Main Function - UDF

Finally Figure 5 shows the main function that represents the UDF that we are trying to process. It calls the Select Function on a table, Join Function on some tables. It then fetches one value from the intermediate tables generated, multiplies them, and then returns the result.

3 Relational and Imperative Optimizations

3.1 Imperative Optimizations

Our IR supports classical imperative optimizations for imperative constructs thanks to the reusable passes of MLIR [6]. As such, we automatically support constant folding/propagation, DCE, LICM, function inlining, etc. However, we can go slightly further than this because we know the distribution of the data. We showcase two examples of how knowledge of data distribution can help imperative optimization.

LICM in Select Statement
Consider the query listed in Figure 6a:

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

(a) Before manual LICM optimization.

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

(b) After manual LICM optimization.

Fig. 6: SQL query with loop invariant code in the projection.

The projection contains loop invariant code that performs expansive computation. We observed that a traditional optimizer consistently fails to extract this computation because the predicate might never pass, which means that code motion could result in unnecessary work. The same call will, therefore, be executed over and over again for every tuple. However, in a DBMS setting, depending on the predicate, we can be confident that the predicate passes for at least one tuple. We can, therefore, manually move the code out of the select statement, and effectively obtain the result shown in Figure 6b.

Dead Code Elimination for Select Statement
Consider the query listed in Figure 7a:

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

(a) Before manual DCE optimization.

```
var result = EMPTY;
```

(b) After manual DCE optimization.

Fig. 7: Dead Select Statement.

While the traditional imperative optimizer is aware that the predicate is false, it is not aware of the fact the table is of finite size. It, therefore, fails to eliminate the iteration through the table to avoid changing the behavior of the program. We, on the other hand, can eliminate the select statement, and prevent a potentially expansive iteration through a large table (shown in Figure 7b).

3.2 True Filters, False Filters

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

(a) True Filter Query - Unoptimized

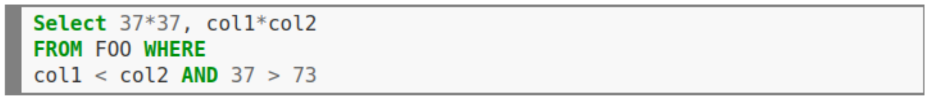
```
Select 37*37, col1*col2
FROM FOO WHERE
col1 < col2
```

(b) True Filter Query - Optimized

Fig. 8: True Filter Query

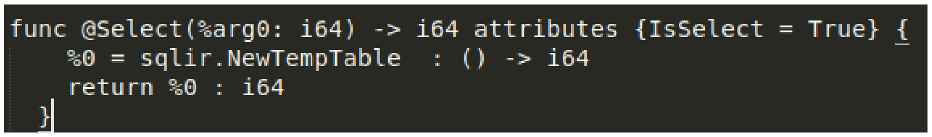
Removal of True and False filters comes very naturally in our optimizations. These filters can be very expensive in a setting where the table size is very large. For eg in Figure 8a is repeatedly called for each tuple in the table. In our optimization passes, this is relatively simple to get done with. The predicate basic blocks whose jump conditions are always true can safely be removed from the SQLIR unless they are calling some functions that alter the state of the system like Date etc. This results in SQLIR equivalent to the query in Figure 9b.
For any false filter we know that we will never return any tuple. Hence the SQLIR gets directly optimized and all of its basic blocks get removed. Hence our temp table gets materialized but nothing gets filled in it. For eg the query shown in Figure 9a after optimization gets converted to the SQLIR in Figure 9. Furthermore our inlining pass also inlines this is in the main function resulting

in almost 0 overhead compared to the case when we were going through the entire table.



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

(a) False Filter Query - Unoptimized



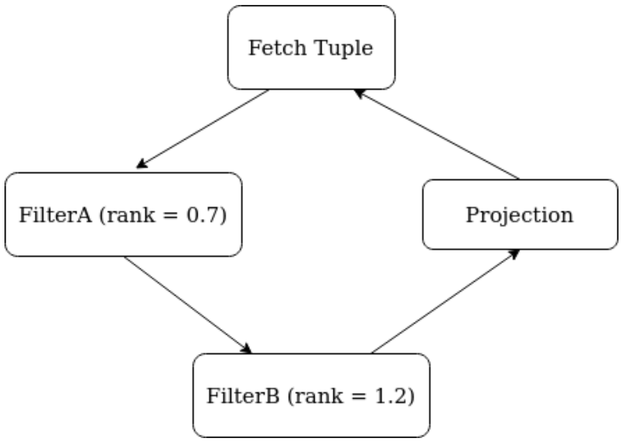
```
func @Select(%arg0: i64) -> i64 attributes {IsSelect = True} {
    %0 = sqlir.NewTempTable  : () -> i64
    return %0 : i64
  }
```

(b) False Filter Query - Optimized

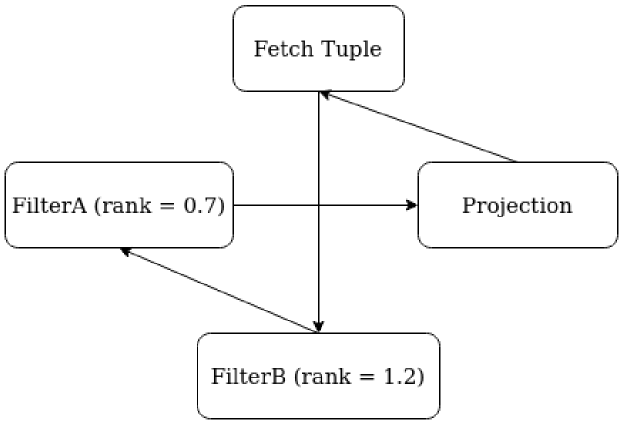
Fig. 9: False Filter Query

3.3 Filter Ordering

Given the structure of our select functions, implementing filter ordering is easy. We need to switch the branch targets depending on the **rank** of the filter. Rank is commonly defined as $\frac{1-s}{c}$ where s is the selectivity and c is cost per tuple [5]. Intuitively, this means that more restrictive filters should be applied first to reduce the number of tuples immediately and that more expansive filters should be applied last to prevent them from operating on too many tuples. For implementation simplicity, we define rank as $\frac{1}{c}$ because computing the selectivity requires techniques like statistics collection on tables that are outside the scope of this project. A visual representation of filter ordering this shown in Figure 11



(a) Branch targets before filter ordering.



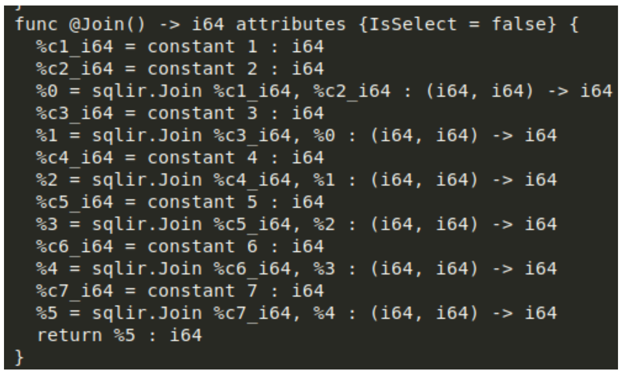
(b) Branch targets after filter ordering.

Fig. 10: Filter Ordering.

3.4 Join Ordering

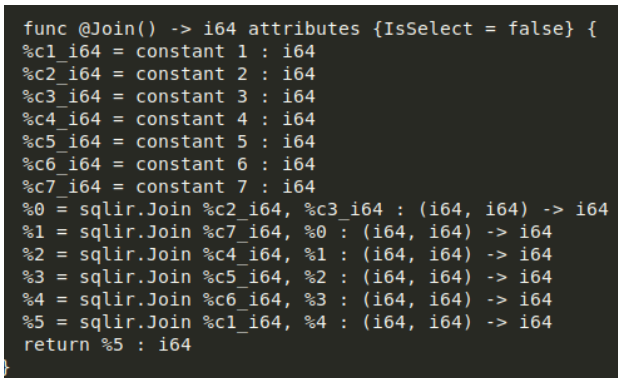
Join Ordering is one of the most important optimizations in the declarative world. Having the correct join ordering can help us differentiate between queries that will never run and queries that can finish in a measure of seconds. A general join ordering framework calculates the cardinality estimate of the output of each intermediate table join and uses it to find the optimal join ordering. This is achieved by doing a bottom-up dynamic programming algorithm similar to the N Matrix Multiplication problem. Including a cost model, cardinality estimates, and statistics in our model were beyond the scope of the project as it requires interfacing with various database internals like storage and optimizer layer. Instead to show that general Join Ordering optimizations are realizable in SQLIR we based our join orderings on the table size. Table size can be used as

a proxy estimate instead of the actual cardinality estimates. The optimization was achieved by writing a pass that appropriately adjusts the parameters of the join operator inside the join function.
Figure 11a represents the input join order and Figure 11b represents the SQLIR after our optimization pass. The join order has been modified. This optimization can be really helpful when merged with statistics and cost model inside the database.



```
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
}
```

(a) SQLIR with Input Join Ordering



```
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
}
```

(b) SQLIR with Modified Join Ordering

Fig. 11: Join Ordering

4 Evaluation

4.1 Experimental Setup

Our setup consists of four phases:

1. UDF definition: We define our UDFs using our custom language inspired by C++ with select statements as an added feature. Rather than implement

a custom parser and type-checker for this language, we hand-code all the UDFs by directly writing the Abstract Syntax Tree.
2. IR generation: We write a pass over this AST to generate SQLIR code.
3. Optimization: We run our custom passes, as well as MLIR's predefined optimizations (inlining, constant folding, etc) on the generated SQLIR.
4. Execution: Ideally, we should lower our SQLIR to LLVM IR [2] and execute it. However, this process is beyond the scope of this project as it has been done before. As a result, we hand code the optimized code using C++ directly for execution.

In short, we implemented SQLIR generation and optimization, and we hand-code UDF definition and execution. We ran all experiments on an Intel Core i5-7300HQ processor (2.5GHz base clock speed, 3.5GHz when boosted). The compiler used is Clang-9 [1].

4.2 Imperative Optimizations

LICM for Select Statements
For this experiment, we execute the query shown in Figure 6, where the expansive call computes the sum of a column in another table of varying size. The results are shown in figure Figure 12. We see that Clang is unable to perform code motion on the expansive function, which results in repeated invocation at every iteration of the loop. Our pass, on the other hand, precomputes the desired sum, which explains the flatness of the curve.

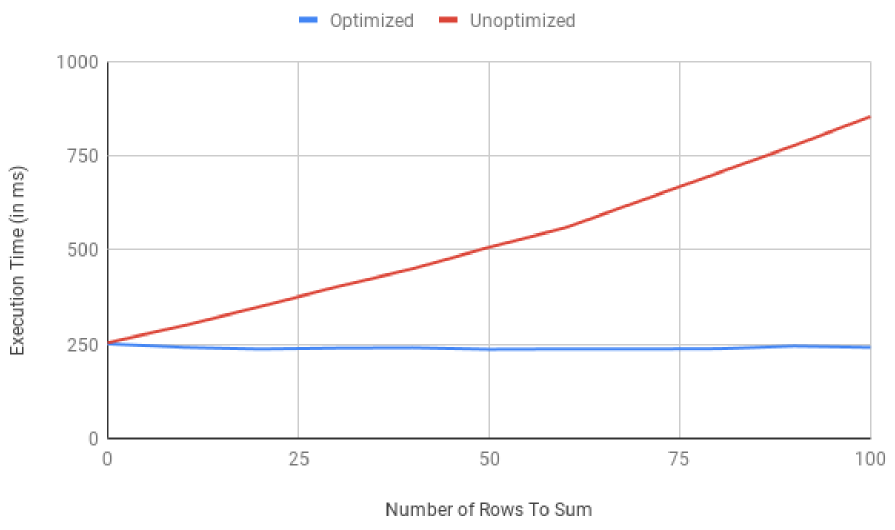


Fig. 12: LICM optimized code versus unoptimized code as the size of the table being summed varies.

DCE for Select Statements
For this experiment, we execute the query shown in Figure 7, where the dead select statement first gets optimized and then gets eliminated. Hence we no longer need to traverse the full table. We ran an experiment to see the difference in performance. The results are summarized in Figure 13. Clearly as the table size increases, the execution time of our query remains constant while this is not true for an SQL optimizer.

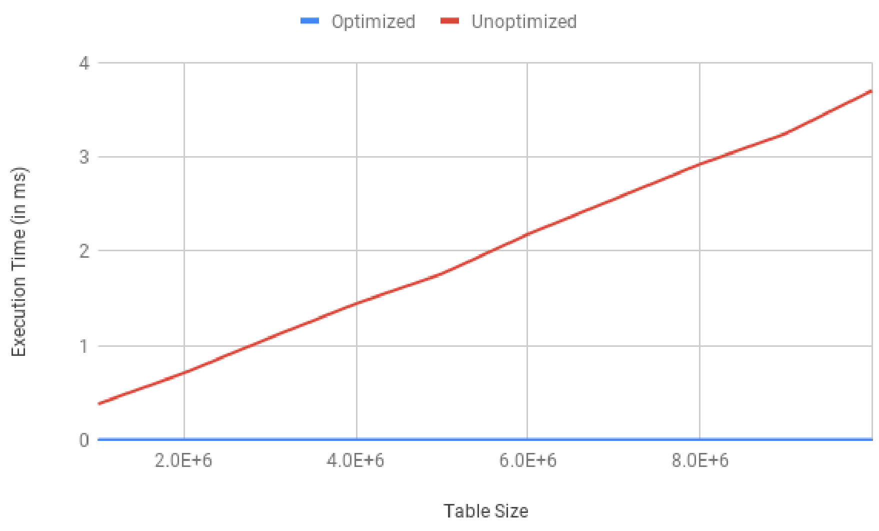


Fig. 13: DCE optimized code versus unoptimized code as the size of the table being scanned varies.

4.3 Filter Ordering

Cost Based Ranking
For this experiment, we use our simplified rank function, where the cost is simply the number of instructions (a more realistic implementation would also use the instruction type). We consider the following query:

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

Fig. 14: Table Scan with an expansive filter and a cheap filter.

Both predicates have the same selectivity. The ordered implementation places the cheap predicate (the one without a modulo) first. The results are show in Figure 15. The unoptimized order has to apply the expansive predicate on all tuples, whereas the optimized order only applies it on a fraction of tuple. As this fraction gets smaller and smaller, the difference between the running time gets larger.

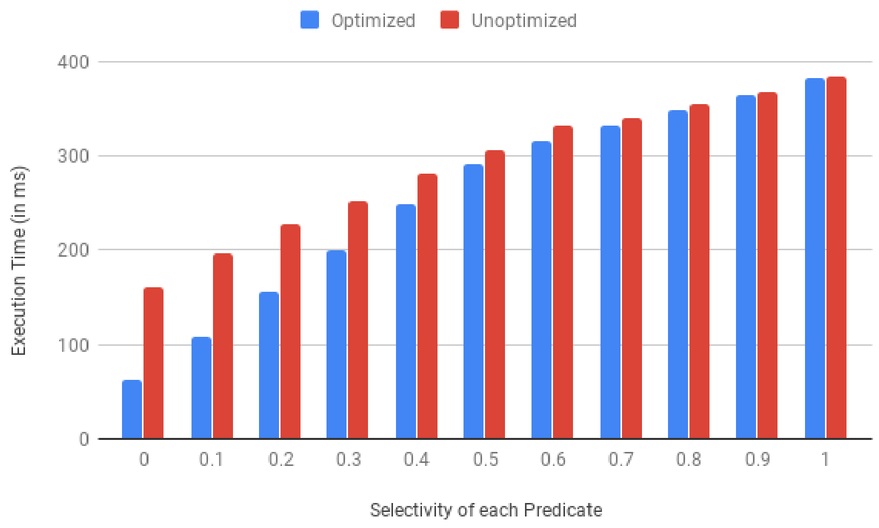


Fig. 15: Effect of cost based ordering of filters as the selectivity varies.

Selectivity Based Ranking
For this experiment, we select filters based on prior knowledge. Our optimization passes have no idea about the selectivities of the filters hence we hardcoded this information in the pass. We consider the following query:

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

Fig. 16: Table Scan with a More Selective Filter and a Less Selective Filter.

Both predicates have the same cost but different selectivities. One of the predicates has a fixed selectivity while the other is being varied in the experiment. The ordered implementation places the varying predicate first. The results are show in Figure 17. When the varying predicate has more selectivity, the unoptimized order has to apply both the filters on most of the tuples while the optimized order only applies the more selective filter most of the time. The anomaly at the starting of the experiment can be attributed to the fact that the varying filter has lesser selectivity than the fixed filter.

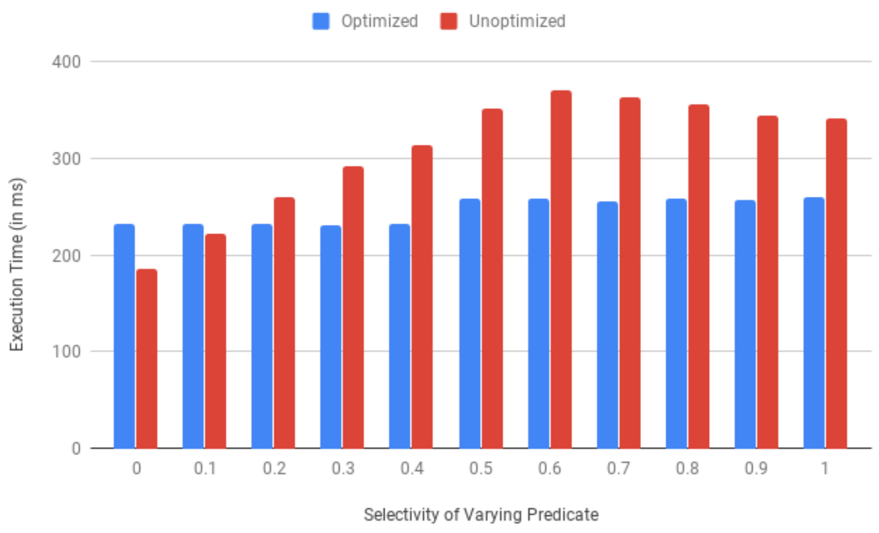


Fig. 17: Effect of selectivity based ordering of filters as the selectivity varies.

4.4 Join Ordering

Three Way Join
In this experiment, we consider the following query:

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

Fig. 18: Three-Way join with a bad initial order. The size of small table is 1000. That of the large table is 10000. That of the medium table varies between a 1000 and 10000.

The unoptimized code keeps the order shown in the query, whereas the optimized code orders tables by size. The result of the experiment is shown in

Figure 19 As a result, the unoptimized code generates a large intermediate result that is much more expensive to compute as the size of the medium table increases. The optimized code is mostly insensitive to the size of the medium table.

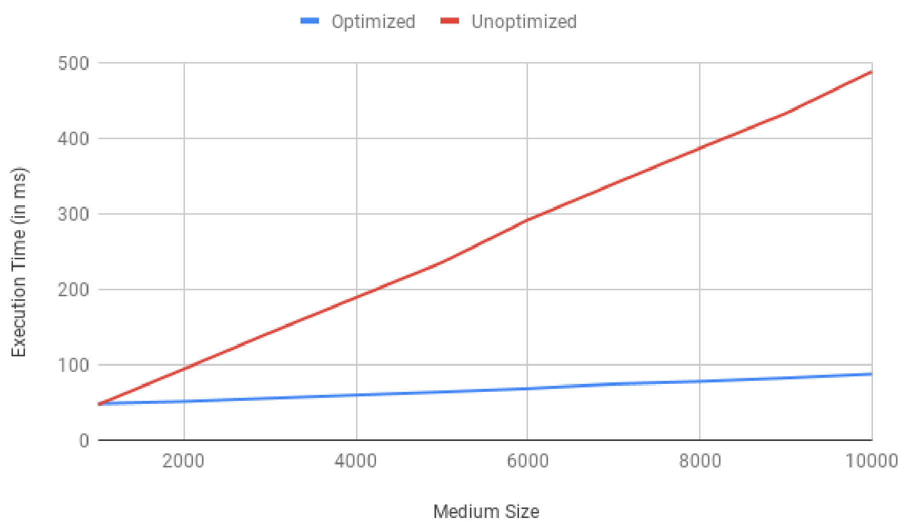


Fig. 19: Effect of size based ordering as the size of medium table varies.

N-Way Join

In this experiment we consider the query in Figure 18 but with a varying number of medium tables. The unoptimized code keeps the order shown in the query while the optimized code orders the table by size and hence results in the order small table, All medium tables, large table. The result is shown in Figure 20. The unoptimized code generates a large intermediate result and this becomes more and more expensive as the number of inner tables increases. The optimized code performs much better. The cost still increases a bit as the number of tables increases as it has to perform more joins but it does not suffer a lot due to the correct join ordering.

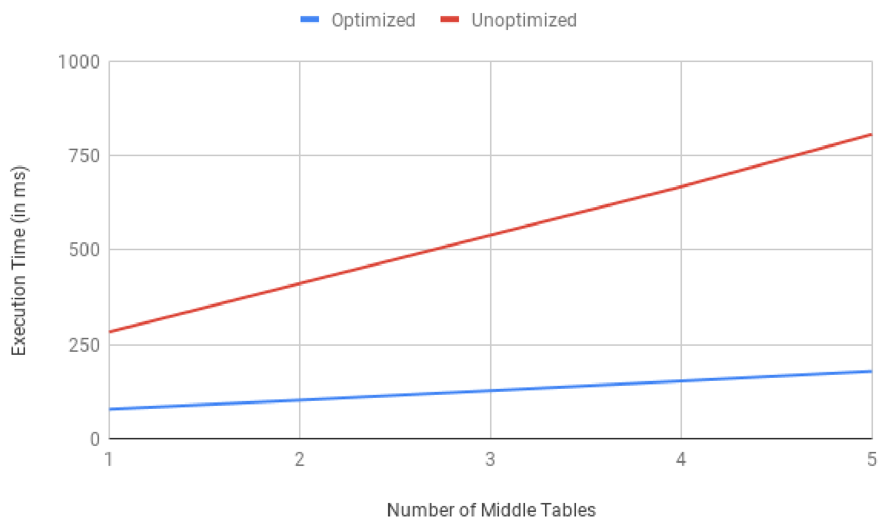


Fig. 20: Effect of size based ordering as number of medium table varies.

5 Surprises and Lessons Learned

The biggest hurdle in our journey was the time it took to get familiar with MLIR. Beyond that, we did not have any big surprises. We did initially try to learn about MLIR through tutorials and talks, which proved mostly unfruitful since they fail to convey anything about what constructs you need to use in practice. The winning strategy was to look at how other domain-specific IRs were implemented and take inspiration from them.

6 Work Contribution

We implemented everything in this project together over Zoom and Google Meet.

7 Conclusion

In this project we endeavored to find a new dynamic optimization paradigm that does not get restricted by the nature of optimizations. This optimization framework aimed to perform optimizations in a general way combining the optimizations found both in the imperative and the declarative world. This is particularly useful when handling UDFs as they lie at the intersection of these two worlds and it's important that we are not restricted while optimizing them. We showed that doing this is indeed possible and built a framework that does both declarative and imperative optimizations. We were also able to leverage some optimizations that only appear when both these worlds combine like DCE for Select Statements while still not losing any optimizations in the declarative world. Being able to Filter and Join Reordering in this world was definitely a win. There appears to be a lot of potential for future work in this direction. As far as we know, no previous work has explored this, and being able to construct this from the ground up was really satisfying.

References

1. Clang: a c language family frontend for llvm. clang.llvm.org
2. The llvm compiler infrastructure. llvm.org
3. Andy Pavlo: 15-721 advanced database systems: Server-side logic. https://15721.courses.cs.cmu.edu/spring2020/slides/24-udfs.pdf (2020)
4. Duta, C., Hirn, D., Grust, T.: Compiling pl/sql away (2019)
5. Hellerstein, J.M., Stonebraker, M.: Predicate migration: Optimizing queries with expensive predicates. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. p. 267–276. SIGMOD '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/170035.170078, https://doi.org/10.1145/170035.170078
6. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: Mlir: A compiler infrastructure for the end of moore's law (2020)
7. Ramachandra, K., Park, K., Emani, K.V., Halverson, A., Galindo-Legaria, C., Cunningham, C.: Optimization of imperative programs in a relational database (2017)