Decorrelation of User Defined Function Invocations in Queries

Varun Simhadri #1, Karthik Ramachandra *2, Arun Chaitanya #3, Ravindra Guravannavar #4, S. Sudarshan *5

# IIT Hyderabad, India
1varun.simhadri@netapp.com, 3arun@worksap.co.jp, 4ravig@acm.org
* IIT Bombay, India
2karthiksr@cse.iitb.ac.in, 5sudarsha@cse.iitb.ac.in

Abstract—Queries containing user-defined functions (UDFs) are widely used, since they allow queries to be written using a mix of imperative language constructs and SQL, thereby increasing the expressive power of SQL; further, they encourage modularity, and make queries easier to understand. However, not much attention has been paid to their optimization, except for simple UDFs without imperative constructs. Queries invoking UDFs with imperative constructs are executed using iterative invocation of the UDFs, leading to poor performance, especially if the UDF contains queries. Such poor execution has been a major deterrent to the wider usage of complex UDFs.

In this paper we present a novel technique to decorrelate UDFs containing imperative constructs, allowing set-oriented execution of queries that invoke UDFs. Our technique allows imperative execution to be modeled using the Apply construct used earlier to model correlated subqueries, and enables transformation rules to be applied subsequently to decorrelate (or inline) UDF bodies. Subquery decorrelation was critical to the wide use of subqueries; our work brings the same benefits to queries that invoke complex UDFs. We have applied our techniques to UDFs running on two commercial database systems, and present results showing up to orders of magnitude improvement.

I. INTRODUCTION

Most database systems provide support for invoking user-defined functions (UDFs) from SQL queries. Calls to UDFs can appear in the SELECT, FROM, WHERE and the HAVING clause of an SQL query. UDFs support a rich set of imperative language constructs such as assignment, conditional branching and loops, and also allow invocation of SQL queries. UDFs encourage modularity, and programmers prefer imperative constructs for many tasks [1]. UDFs also make it possible to express certain tasks that are hard or impossible to write in standard SQL.

Most database systems today inline simple, single-statement UDFs into the query that invokes them. However, to the best of our knowledge, they treat complex UDFs with imperative constructs as black boxes. When a UDF appears in the SELECT or the WHERE clause, it is invoked for every tuple produced by the FROM clause, after the application of simple predicates.

Example 1 shows a query which invokes a UDF in its SELECT clause. The UDF returns the service level for a given

```
CREATE FUNCTION service_level(c custkey) RETURNS char(10) AS
BEGIN
    float totalbusiness; string level;
    select sum(totalprice) into :totalbusiness
    from orders where custkey=:ckey;
    if (totalbusiness > 1000000)
        level = 'Platinum';
    else if (totalbusiness > 500000)
        level = 'Gold';
    else
        level = 'Regular';
    return level;
END
```

Example 2 Decorrelated Form of Query in Example 1
```
SELECT c.custkey, case when e.totalbusiness > 1000000 then 'Platinum'
                          when e.totalbusiness > 500000 then 'Gold'
                          default 'Regular'
                    end as service_level
FROM customer c LEFT OUTER JOIN e ON c.custkey=e.custkey;
```

where e stands for the query

```
SELECT custkey, sum(totalprice) as totalbusiness
FROM orders GROUP BY custkey;
```

customer. It executes a scalar SQL query to compute the customer’s total business, which then uses to decide the service level in a nested if-then-else block. The execution plan for queries such as the one in Example 1 on a commercial database system, is to invoke the UDF for each tuple. Such iterative plans can be very inefficient, since queries within the function body may be executed multiple times, once for each outer tuple.

These plans can be compared to correlated execution of parameterized nested subqueries. In the case of nested subqueries, decorrelation techniques have been well studied [2], [3], [4], [5], [6]. The Apply operator was introduced by Galindo-Legaria et al. [5] to explicitly model correlated execution of subqueries; [5] also presented transformation rules that can replace Apply operators by standard relational operations such as joins, under certain conditions, thereby decorrelating the query. Query decorrelation enables set-oriented execution
plans by rewriting a nested query as a flat query. Once a query is decorrelated, the query optimizer can consider alternative join algorithms such as hash-join and merge-join, in addition to nested loops join.

However, decorrelating UDF invocations, such as the one in Example 1 is a more complex task due to the presence of various imperative constructs. Example 2 shows the same query after decorrelation of the UDF invocation. This transformed query enables set-oriented execution plans, thereby expanding the space of alternative plans for an optimizer. Such transformations have not been addressed so far, to the best of our knowledge. In general, it may not be possible to completely decorrelate queries containing all types of UDFs, but as we show in this paper, there exists a large class of UDFs that can be decorrelated.

In this paper, we extend existing query decorrelation techniques to decorrelate invocations of complex side effect free UDFs with imperative constructs. In particular, we make use of the Apply operator introduced in Galindo-Legaria et al. [5] to decorrelate subqueries. We extend the Apply operator to model UDF invocation and execution, and use transformation rules given in [5], as well as new ones that we propose, to decorrelate UDF invocations.

Our contributions in this paper are as follows:

- We show how queries containing UDFs can be given an algebraic representation. Using our extensions to the Apply operator, we show how UDFs with various imperative constructs such as assignments, conditional branching, and loops can be modeled as expressions.
- We propose a set of transformation rules that can be used in a cost-based optimizer to perform decorrelation. These transformation rules work with existing rules for query decorrelation, and enable the application of already known rules.
- Our techniques are applicable to a large class of UDFs seen in practice. While the rules can be used in a cost-based optimizer, in the absence of access to a cost-based optimizer, they can also be used to implement a query rewriting system outside of the database. We have built such a query rewriting tool to show the effectiveness of our techniques.
- We have conducted a detailed experimental evaluation of our techniques by performing query rewriting, and executing original and rewritten queries on two commercial database systems. Our experiments show an order of magnitude improvement in performance due to decorrelation of UDF invocations.

The paper is organized as follows. We start with an overview of our approach in Section II. We define the terminology used in this paper in Section III. Our technique is described in detail in Section IV, Section V, and Section VI. We consider UDFs with loops in Section VII. We discuss related work in Section VIII. In Section IX, we describe our implementation, present our experimental results in Section X and conclude in Section XI.
optimal, and introduce additional transformations. In particular, decorrelation is cost-based, since correlated evaluation remains as an alternative for the optimizer to consider.

In this paper, we show how the Apply operator can be extended to model UDF invocations and imperative constructs. We consider UDFs with imperative statements such as variable assignment, conditional branching and loops. Our approach for decorrelating UDF invocations involves the following steps:

1) Construct a parameterized expression tree corresponding to the UDF, using the extended Apply operators. The expression is parameterized in the sense that it uses variables whose values are assigned by the calling context. The calling context may be a query or another UDF.

2) Merge the expression tree constructed for the UDF and the expression tree of the outer query block. During this step the parameters for the UDF expression are bound to the attributes of the calling context.

3) Remove the Apply operators (and thus perform decorrelation) using the transformation rules that we present in Section V, as well as known transformation rules such as those presented in [5].

We currently consider side effect free UDFs written in SQL with procedural extensions. However, our work can be extended to UDFs written in other languages, which is an area of future work. Our techniques can be used with scalar and table valued UDFs; however, updates and deletes to table valued attributes are not handled currently.

III. TERMINOLOGY

Before we describe our approach to decorrelate UDF invocations, we define the terminology used in this paper.

1) **The Single relation (S):** This is a relation with a single empty tuple and no attributes. It is used to return scalar constants or computed values as relations.

2) \( \perp \): Value of an uninitialized variable. It can be either null or a language specific default value for the data type.

3) \( \Pi^2 \): Projection without duplicate removal.

4) **Conditional Expressions:** We denote conditional expressions using the following notation:
   \[
   (p_1 ? e_1 : p_2 ? e_2 : \ldots : e_n)
   \]
   An expression of this form evaluates to \( e_1 \) if predicate \( p_1 \) evaluates to true, to \( e_2 \) if \( p_2 \) evaluates to true and so on. If none of the predicates \( p_1, \ldots, p_{n-1} \) evaluates to true, the expression evaluates to \( e_n \). The SQL case statement is analogous to a conditional expression, and is a convenient way to compute an expression in a predicated manner.

5) **Generalized projection** [7]: Projection (both with and without duplicate removal) can involve expressions. The result of an expression \( e \) can be assigned a new name \( n \), using the syntax \( e \) as \( n \). Note that the expression can invoke a UDF and can also be a conditional expression as described above.

6) **Rename operator** [7]: \( \rho(r(a_1, \ldots, a_n))(e) \) returns the result of relational algebra expression \( e \) under the name \( r \) with attributes renamed as \( a_1, \ldots, a_n \). When only attribute renaming is needed we use \( \rho^r(a_1, \ldots, a_n)(e) \). Individual attributes can also be renamed using the as keyword.

7) **Group-by operator:** \( a_1, \ldots, a_n \sum f_1(\ldots f_m() \} (e) \) is used to denote a group-by expression, where \( a_1, \ldots, a_n \) are the grouping columns, and \( f_1, \ldots, f_m \) are the aggregate functions. Grouping columns are optional.

As mentioned earlier, we extend the Apply operator to model imperative constructs in the UDF. We define three extensions of the standard Apply operator.

1) **Apply-Bind extension:** UDF invocations implicitly map formal parameters to actual parameters. In order to represent UDF invocations algebraically, we define a bind extension to the Apply operator. This extension allows the Apply operator to optionally accept a list of parameter mappings of the form \( p_1 = a_1, \ldots, p_n = a_n \), where \( a_1, \ldots, a_n \) are the attributes of the left child of the Apply, and the right child is parameterized by \( p_1, \ldots, p_n \). Such a mapping, if provided, is performed by the operator before evaluating its right child. We denote this as follows:
   \[
   E_1 \mathcal{A}^{\text{bind}}_{p_1 = a_1, \ldots, p_n = a_n} E_2(p_1, \ldots, p_n)
   \]

2) **Apply-Merge extension** (\( A^M \)): This is used to model assignment statements. The right child of the apply operator computes the values for attributes which are then assigned to (or merged with) the attributes present in the left child. Let \( r \) be a relation with schema \( R = (a_1, \ldots, a_n) \). Let \( e(r) \) be a parameterized single-tuple expression, whose result has the schema \( S = (b_1, \ldots, b_m) \). Let \( L \) be a sequence of assignments of the form \( a_1 = b_1, a_2 = b_2, \ldots, a_k = b_k \). Now, Apply-Merge \( r \mathcal{A}^M(r) e(r) \) can be defined procedurally as follows: For each tuple \( t \in r \), evaluate \( s = e(t) \). Then produce \( t' \) as an output tuple, where \( t' \) is obtained from \( t \) after performing the assignments specified in \( L \). We can define the operation algebraically as follows:
   \[
   \rho (X)(r \mathcal{A}^M L e(r))
   \]

   where \( X = r \times \{ a_1, \ldots, a_k \} \), \( b_1 \) as \( a_1, \ldots, b_k \) as \( a_k \).

   The assignment list \( L \) is optional. When, omitted it is assumed to be of the form \( r.c_1 = s.c_1, \ldots, r.c_k = s.c_k \), where \( c_1, \ldots, c_k \) are the attributes common to \( R \) and \( S \). Note that the above definition assumes \( e(r) \) to be exactly one tuple. If the r.h.s expression of an assignment statement results in more than one tuple, an exception is thrown. If it is empty, then it may either throw an exception, or perform no assignment and retain the existing value. The semantics of assignment statements when \( e(r) \) is empty, or has more than one tuple, varies across systems, and modeling it is part of our future work.

3) **Conditional Apply-Merge operator** (\( \mathcal{A}^M_c \)): Let \( r \) be a relation with schema \( R = (a_1, \ldots, a_n) \). Let \( e_1(r) \) and \( e_f(r) \) be parameterized single-tuple expressions, and \( p(r) \) be a parameterized predicate expression. Now, the Conditional Apply-Merge operator is defined as follows:
   \[
   r \mathcal{A}^M_c(p(r), e_1(r), e_f(r)) = r \mathcal{A}^M (\sigma_{p(r)}(e_1(r))) \cup \sigma_{\neg p(r)}(e_f(r)))
   \]

   This operator is used to model assignments within if-then-else blocks in the body of a UDF; more details and an example are given in Section IV.

With this background, we now proceed to describe our approach of UDF decorrelation.
Example 3 UDF with a Single Arithmetic Expression

```
create function discount(float amount) returns float as begin
    return amount * 0.15;
end
```

**Query:** select orderkey, discount(totalprice) from order;

Fig. 2. Expression Tree for the UDF in Example 3

IV. ALGEBRAIC REPRESENTATION OF UDFs

The first step towards decorrelating a UDF invocation is to construct a parameterized algebraic expression corresponding to the UDF. This expression is later merged with the expression tree of the calling query or function.

Consider the UDF of Example 3, which contains a single statement returning the value of an arithmetic expression. The expression tree constructed for the UDF of Example 3 is shown in Figure 2. This tree has one `Apply` operation whose left child is the `Single` relation, and the right child is a projection on the `Single` relation that computes the arithmetic expression. Finally, there is a projection on the return value.

Obviously, this expression is not in its simplest form, and can be simplified. But it shows a way to express scalar computations as relational expressions, and we show how this generalizes to any statement in the body of a UDF.

Similar expression trees can be constructed for statements with different kinds of expressions (logical, relational) and datatypes. As another example, consider the query of Example 4, in which the UDF contains a single parameterized query execution statement. The expression constructed for the UDF is shown in Figure 3. We note that many commercial database systems inline single statement UDFs such as Example 3 and Example 4 and optimize them. However, we have considered these examples to illustrate our technique of building algebraic expressions for statements in UDFs.

We now propose a general technique to algebraize arbitrary side effect free UDFs with conditional branching and other imperative constructs. In this section we handle UDFs without loops, and describe our approach to handle UDFs with loops in Section VII.

We use the control flow graph (CFG) [8], a commonly used program representation, to explicitly capture control flow through the statements of a function. Each node in a CFG corresponds to a statement in the UDF. A directed edge between two nodes represents control flow. The CFG has a start node, from which execution begins, and an end node where execution terminates.

Example 4 UDF with a single SQL query

```
create function totalbusiness(int ckey) returns int as begin
    return select sum(totalprice) from orders
    where custkey=:ckey;
end
```

**Query:** select custkey, totalbusiness(custkey) from customer;

Fig. 3. Expression Tree for the UDF in Example 4

In order to suit our needs, if-then-else blocks are logically treated as single nodes. Nested if-then-else blocks are treated as nested logical nodes. The CFG for the UDF in Example 1 is shown in Figure 4 with nodes labeled \(N_1, \ldots, N_8\). The logical nodes with nested if-then-else blocks are labeled \(L_0, \ldots, L_4\), and shown in dashed boxes where applicable. The logical block \(L_3\) has two nested logical blocks denoted as \(L_{3.1}\) and \(L_{3.2}\). As it can be seen, the resulting graph (considering top-level logical nodes) would have no branching.

Each node \(N_i\) in the CFG contributes to the expression tree. The contribution of node \(N_i\) is denoted by \(E_{N_i}\), and the contribution of a logical node \(L_i\) is denoted by \(E_{L_i}\). These contributions are computed as follows.

\[
E_{L_i} = \begin{cases} 
S & \text{if } L_i \text{ is Start} \\
\Pi_{r \rightarrow s}(S) & \text{if } L_i \text{ is an assignment } l = r \\
(p, e_t, e_f) & \text{if } L_i \text{ is an if-then-else block}
\end{cases}
\]

The expression \(E_{L_0}\) for the start node is the `Single` relation. An assignment statement of the form \(l = r\) is represented as a generalized projection on the `Single` relation. Note that \(r\) can be a program expression, a scalar SQL query, or a UDF invocation. If \(r\) is a scalar SQL query, we use its relational expression. If it is a UDF invocation, we first build an expression for the called UDF, and then use it in the projection. If an algebraic representation cannot be built for the called UDF, it is left as a function invocation. Variable declarations are treated as assignments with the r.h.s. as \(\bot\), i.e., the default uninitialized value for the datatype.

An if-then-else block has two successors corresponding to the then and the else parts. In this case, we recursively define the contribution of the if-then-else block as the set of expressions \((p, e_t, e_f)\). Here \(p\) is the predicate of the if statement, \(e_t\) is the expression tree corresponding to the then branch (i.e., \(p\) is true), and \(e_f\) is the expression tree corresponding to the else branch (i.e., \(p\) is false). This expressions \((p, e_t, e_f)\) captures the contribution of the entire conditional block and hence the block can be logically seen as a single node in the CFG. All
the contributions of individual nodes are then combined to get $E_{udf}$, the expression tree for the UDF as shown below.

$$E_{udf} = E_{L_0}$$

for $i$ from 1, ..., $k$ do // $k$ is the # of logical nodes
  $o_i = \text{chooseApplyType}(L_i)$
  $E_{udf} = E_{udf} \cdot \cdot L_i$
end

$E_{udf}$ is initially assigned to $E_{L_0}$ corresponding to Start. Then, for every successive logical node $L_i$, we add an Apply operator whose left child is the expression built so far ($E_{udf}$), and right child is $E_{L_i}$.

The Apply operator’s type $o_i$ depends on the corresponding node in the UDF. Variable declarations use the Apply-cross($A^\times$) operator. Assignment of values to previously defined local variables is algebraized using Apply-Merge($A^M$). The assignment of results of a scalar query to scalar variables also uses Apply-Merge. Conditional branching nodes (i.e. if-then-else blocks) use the Conditional Apply-Merge ($A^{\times}_{C}$) operator. The return clause is mapped to an apply-cross ($A^\times$) with a relational expression corresponding to the return expression. As a convention, we always alias the return value to the name retval. Finally, a projection on retval is added to complete the expression for the UDF.

We now illustrate the construction of the expression tree for the CFG in Figure 4. The expressions with their corresponding Apply operator types are as follows:

- $e_{L_1} = \Pi_0 \text{ as totalbusiness, null as level}(S)$
- $A^{o_1} = A^\times$
- $e_{L_2} = \pi_v(\text{sum(totalprice}) \text{ as v(custkey} = ckey(\text{orders}))$
- $A^{o_2} = A^M_{\text{totalbusiness}}$
- $e_{L_3} = \text{totalbusiness} > 1000000, e_{L_{3.1}}, e_{L_{3.2}}$
- $A^{o_3} = A^M_{C} (\text{totalbusiness} > 1000000)$

Since $L_3$ is an if-then-else block, the expression $e_{L_3}$ is defined recursively in terms of the predicate at $N_3$, the true branch at $L_{3.1}$ and the false branch at $L_{3.2}$. The conditional-apply-merge operator is used. Since $L_{3.1}$ comprises of a single node $N_4$, its expression would be $A^M_{C}$ $e_{N_4}$ which is the same as $e_{N_4}$. In general, the expression for any logical node $L_i$ that comprises of a single node $N_j$ would be $e_{N_j}$ (see rule R1 of Table II). The expression for $L_{3.2}$ which is another if-then-else block, is defined in terms of $e_{N_6}$ and $e_{N_7}$. The remaining expressions and operator types are as below:

- $e_{L_{3.1}} = e_{N_4} = \Pi_{\text{Platinum}}' \text{ as level}(S)$
- $e_{L_{3.2}} = (\text{totalbusiness} > 500000, e_{N_6}, e_{N_7})$
- $A^{o_{3.2}} = A^M_{C} (\text{totalbusiness} > 500000)$
- $e_{N_6} = \Pi_{\text{Gold}}' \text{ as level}(S)$
- $e_{N_7} = \Pi_{\text{Regular}}' \text{ as level}(S)$
- $e_{L_4} = \Pi_{\text{level as retval}(S)}$
- $A^{o_4} = A^\times$

Using these expressions and Apply operators, we construct the tree as described. The resulting tree for the UDF in Figure 4 is shown in Figure 5. For clarity, the tree shows the predicate at each conditional-apply-merge operation. This tree is further simplified while removing the Apply operators.

V. EXPRESSION TREE Merging

Once the expression tree is constructed for the UDF, it needs to be correlated with the query that invokes the UDF. This is very similar to the way nested subqueries are correlated with the outer query except for one key difference: the formal parameters of the UDF have to be bound to their corresponding actual parameters produced by each tuple of the outer query block. To this end, we make use of the enhanced Apply operation (Apply with the bind extension) defined in Section III to merge the expression tree of the outer query with the tree constructed for the UDF.

Let $E_{outer}$ and $E_{udf}$ be the expression trees corresponding to the outer query block and the UDF respectively. Also,
let \( f_{p1}, \ldots, f_{pn} \) denote the formal parameters of the UDF, and \( a_1, \ldots, a_n \) denote attributes of \( E_{\text{outer}} \) that are the actual parameters to the UDF. Irrespective of whether the UDF invocation is in the \textit{where} clause or the \textit{select} clause, \( E_{\text{outer}} \) will have the UDF invocation as the right child of an \textit{Apply} operation [5] (as shown in the LHS of Figure 6). Now, we merge them as follows:

1) The UDF invocation is replaced by its algebraic form \((E_{\text{udf}})\) as the right child of the \textit{Apply} operator. In Figure 6, the invocation \( func(p_1, \ldots, p_n) \) is replaced by its algebraic form \( E_{func} \). The expression \( E_{func} \) is parameterized by formal arguments \( f_{p1}, \ldots, f_{pn} \).

2) The list of parameter mappings of the form \( f_{p1} = a_1, \ldots, f_{pn} = a_n \) is passed to the \textit{Apply} operator (with the \textit{bind} extension) as illustrated in Figure 6. These assignments are performed by the \textit{Apply} operator before evaluating its right child (\( E_{\text{udf}} \)).

We have now constructed a merged expression tree for the UDF and its calling query block. In the next section, we describe how to remove the \textit{Apply} operators and simplify this tree.

VI. REMOVAL OF APPLY OPERATORS

The \textit{Apply} operators present in the merged query tree are removed using the equivalence rules given by Galindo-Legaria et al. [5], and additional equivalence rules presented in this paper. For completeness, we show some of the known equivalence rules (rules K1–K6) in Table I.

Equivalence rules (R1 – R9) given in Table II are required in order to express extended \textit{Apply} operations in terms of standard \textit{Apply} or other relational operations. This enables application of known rules, thereby simplifying and decorrelating the expression. We now briefly describe rules R1 – R9. Let \( r \) be a relation with schema \( R(a_1, \ldots, a_n) \).

Rule R1: This rule removes the \textit{Apply-cross} operator when one of its children is \textit{Single}.

\[
r \odot S = S \odot r = r
\]
i.e., if one of the children of an \( \odot \) is the \textit{Single} relation and the other child is \( r \), the result of the operation is \( r \).

Rule R2: This rule enables the removal of \textit{Apply-merge} when its right child is a projection on \textit{Single}.

\[
r \odot \Pi_A(S) = \Pi_B(r)
\]
where \( A = (e_1 \text{ as } a_1, \ldots, e_k \text{ as } a_k) \) and \( B = R - \{a_1, \ldots, a_k\} \), i.e., if the inner expression of an \( \odot \) is a projection on \textit{Single}, then the operation can be written as a projection on \( r \) with common attributes being projected as from \textit{Single}.

Rule R3: The function composition rule for the generalized projection operator where \( f \) and \( g \) are pure functions:

\[
\Pi_f(B)(\Pi_g(A) \text{ as } B(r)) = \Pi_f(g(A)(r))
\]

Rule R4: \textit{Apply-merge} removal. This rule follows from the definition of \textit{Apply-merge} (Section III).

Rule R5: Move a projection after the \textit{Apply}.

\[
(\Pi_A(r) \circ \odot e) = \Pi_A(\circ \odot e)
\]

Rule R6: Conditional-\textit{Apply} removal. This rule follows from the definition of \textit{Conditional-Apply} (Section III).

Rule R7: Union to generalized projection. A union between expressions with mutually exclusive selection predicates can be written as a generalized projection with a conditional expression.

\[
\Pi_{c_1} \text{ as } a(\sigma_{p_1}(r)) \cup \Pi_{c_2} \text{ as } a(\sigma_{p_2}(r)) = \Pi_{c_1 \vee c_2}(\sigma_{p_1 \vee p_2}(r)) = a(r)
\]

Rule R8: This rule can be derived from rules R6 and R7 to express a \textit{Conditional-Apply} as projection directly, whenever \( c_1(r) \) and \( c_2(r) \) are scalar valued.

Rule R9: \textit{Apply-bind} removal. An \textit{Apply} operation with bind extension can be removed by replacing all occurrences of formal parameters \( (p_1, \ldots, p_n) \) in its right child by actual parameters \( (a_1, \ldots, a_n) \).

\[
r \odot \Pi_{bind} p_1 = a_1, \ldots, p_n = a_n e(p_1, \ldots, p_n) = r \odot e(a_1, \ldots, a_n)
\]

Using the above rules R1 – R9, we now illustrate the construction of the merged query tree and removal of \textit{Apply} operators for the examples we have been considering so far. First, let us consider the query in Example 3. For this example, \( E_{\text{outer}} \) is given by:

\[
E_{\text{outer}} = \Pi_{\text{orderkey}, \text{discount(totalprice) as } d(\text{order})}
\]

This can be written using the \textit{Apply} operator (rule K6) as:

\[
E_{\text{outer}} = \Pi_{\text{orderkey}, \text{d(orders \odot \text{orderkey, discount(totalprice)})}}
\]

The expression for the UDF, \( E_{\text{udf}} \) is shown in Figure 2, and the corresponding merged query tree is shown in Figure 7. This merged expression can now be simplified and the \textit{Apply} operators removed by using rules in [5] (Table I) and Table II. Applying rule K4 and R1 for the innermost \textit{Apply} operator, we get the expression:

\[
\Pi_{\text{orderkey, retval as } d(\text{orders \odot a(\text{amount(totalprice)}) \Pi_{\text{amount} = 0.15 as retval(S)})}}
\]

Note that the above expression still uses the formal argument \textit{amount}, which is replaced by the actual argument when the


<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>$r \ A \ e = r \otimes_{true} e$, if $e$ uses no parameters from $r$</td>
</tr>
<tr>
<td>K2</td>
<td>$r \ A \otimes (\sigma_p(e)) = r \otimes_{p} e$, if $e$ uses no parameters from $r$</td>
</tr>
<tr>
<td>K3</td>
<td>$r \ A^x (\sigma_p(e)) = \sigma_p(r \ A^x e)$</td>
</tr>
<tr>
<td>K4</td>
<td>$r \ A^x (\Pi_v(e)) = \Pi_{v \cup \text{schema}(r)}(r \ A^x e)$</td>
</tr>
<tr>
<td>K5</td>
<td>$r \ A^x (\mathcal{G}_F(e)) = A_v \ 	ext{schema}(G_F(r \ A^x e)$</td>
</tr>
<tr>
<td>K6</td>
<td>$\Pi_{f(A)} as a_1, a_2, \ldots, a_n(r) = \Pi_{a_0, a_1, \ldots, a_n}(r \ A^x \rho_{a_0}(f(A)))$</td>
</tr>
</tbody>
</table>

**TABLE I. KNOWN RULES FOR CORRELATION REMOVAL [5]**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>$r \ A^x \ S = S \ A^x r$</td>
</tr>
<tr>
<td>R2</td>
<td>$r \ A^M(\Pi_{a_1 as e_1, \ldots, a_k as e_k}(S)) = \Pi_{A, e_1 as a_1, \ldots, e_k as a_k}(r)$</td>
</tr>
<tr>
<td>R3</td>
<td>$\Pi_{f(B)}(\Pi_{g(A)} as B(r)) = \Pi_{f(g(A))}(r)$</td>
</tr>
<tr>
<td>R4</td>
<td>$r \ A^M(L) e(r) = \Pi_X(r \ A^x e(r))$</td>
</tr>
<tr>
<td>R5</td>
<td>$(\Pi_{d_1 as a_1, \ldots, a_k as e_1 as b_1 as v_1, \ldots, e_m as b_m as v_m}(r)) A^\otimes e = \Pi_{a_1, \ldots, a_k, e_1 as b_1, \ldots, e_m as b_m, \ldots} (r \ A^\otimes e)$</td>
</tr>
<tr>
<td>R6</td>
<td>$r \ A^M(p(r), e_1(r), e_2(r)) = r \ A^M(\sigma_{p_1}(e_1) \sigma_{p_2}(e_2))$</td>
</tr>
<tr>
<td>R7</td>
<td>$\Pi_{e_1 as a_1, \ldots, a_k as e_1, e_2 as a_1, \ldots, a_k as e_2}(r)$</td>
</tr>
<tr>
<td>R8</td>
<td>$r \ A^M(p(r), e_1(r), e_2(r)) = \Pi_{r, e_1, e_2}(r)$</td>
</tr>
<tr>
<td>R9</td>
<td>$r \ A^\otimes_{bind: p_1=a_1, \ldots, p_n=a_n, e_1 as b_1, \ldots, b_n}(r) = r \ A^\otimes e(a_1, \ldots, a_n)$</td>
</tr>
</tbody>
</table>

**TABLE II. ADDITIONAL EQUIVALENCE RULES**

**Fig. 7. Merged Expression Tree for the Query in Example 3**

Apply operator (with bind extension) is removed (rule R9). Applying rule K4 and R1 again, we get the final expression:

$$\Pi_{\text{orderkey, totalprice}^{0.15} \text{ as d}(\text{orders})}$$

Queries that invoke a UDF in their WHERE clause can be handled in a similar manner. For example, consider the following query that invokes the same UDF of Example 3 in its WHERE clause:

**select orderkey from orders where discount(totalprice) > 100;**

This query is initially represented as follows:

$$E_{\text{outer}} = \Pi_{\text{orderkey}}(\sigma_{d>100}(\text{orders} \ A^x \rho_d(\text{discount(totalprice)})))$$

Merging this with the expression for the UDF we get:

$$\Pi_{\text{orderkey}}(\sigma_{d>100}(\rho_d(e)))$$

where $e$ is the expression in Figure 7. After simplification, the final expression is:

$$\Pi_{\text{orderkey}}(\sigma_{d>100}(\rho_d(\text{e})))$$

Consider the query of Example 4. The query is initially represented as:

$$E_{\text{outer}} = \Pi_{\text{customer} A^x \rho_v(\text{totalbusiness(custkey))}}$$
The expression for the UDF is shown in Figure 3. Merging them, we get:

$$\Pi_{\text{custkey}}\, \nu(\text{customer} \times \rho_c(\text{service}_n \times e))$$

where

$$e = \Pi_{\text{selval}}(\sigma_{\text{sum(totalprice)}} \text{ as retval}(\sigma_{\text{custkey}=\text{ckey}}(\text{orders})))$$

Applying rule R1 and simplifying, we get,

$$\Pi_{\text{custkey}}\, \nu(\text{customer} \times \text{sum(totalprice)} \text{ as } \nu(\sigma_{\text{custkey}=\text{ckey}}(\text{orders})))$$

Transformations proposed in [5] can then be used to remove the correlation and obtain the following expression as one of the equivalent forms.

$$\Pi_{\text{custkey}}\, \nu(\text{customer} \hat{=} \lambda d(\text{custkey} \times \text{sum(totalprice)} \text{ as } \nu(\text{orders})))$$

Consider the UDF and query of Example 1. The query is initially represented as:

$$E_{\text{outer}} = \Pi_{\text{custkey}}\, \nu(\text{customer} \times \rho_c(\text{service}_n(\text{custkey})))$$

The parameterized expression $$E_{\text{udf}}$$ constructed for the UDFs shown in Figure 5. Let $$p_1$$ and $$p_2$$ be the predicates ($$\text{totalbusiness} > 1000000$$) and ($$\text{totalbusiness} > 500000$$) respectively. After applying rule R1 on $$E_{\text{udf}}$$, we get,

$$E_{\text{udf}} = \Pi_{\text{level}}\, \nu(\text{retval}(\Pi_{\text{customer}}(\text{udf}(p_1, e_{\text{N}_1}, e_{\text{T}_2})))$$

where

$$\text{T}_1 = \text{e}_{\text{L}_1} \times \text{M}_{\text{C}}(\text{M}_{\text{P}_2}(\text{e}_{\text{N}_0}, e_{\text{N}_7})).$$

Using rule R4, K4 and R1. $$\text{T}_1$$ can be simplified to:

$$\text{T}_1 = \Pi_{\text{totalbusiness}}, \text{null as level}(\text{e}_{\text{L}_2})$$

Using rule R8, we get: $$\text{T}_2 = \Pi_{\text{p}_2=\text{e}_{\text{N}_0}, e_{\text{N}_7}}(\Pi_{\text{S}})$$ Merging $$E_{\text{udf}}$$ with $$E_{\text{outer}}$$ and simplifying with rule K4, we get,

$$\Pi_{\text{custkey}}\, \nu(\Pi_{\text{totalbusiness}}, \text{null as } \nu(\text{e}_{\text{L}_3} \times \text{M}_{\text{C}}(\text{p}_1, e_{\text{N}_4}, e_{\text{T}_2})))$$

where

$$\text{T}_3 = \text{(customer} \times \text{e}_{\text{L}_2}).$$

Using the transformations in [5], we get:

$$\text{T}_3 = \text{customer} \times \text{M}_{\text{C}}(\text{custkey} \times \text{sum(totalprice)} \text{ as } \text{totalbusiness}\times \text{(orders})))$$

Applying rule R8 to the merged expression, we get the following final simplified expression. The SQL query corresponding to this expression is given in Example 2:

$$\Pi_{\text{custkey}, \text{p}_1=\text{Platinum'}, \text{p}_2=\text{Gold'}, \text{Regular'}} \times \nu(\text{e}_{\text{T}_3})$$

VII. UDFs with Loops

Loops are encountered quite often in UDFs, and loops that iterate over cursors defined on query results are common. Example 5 shows a query on the TCP-H schema which invokes the UDF totalloss, with a cursor loop in it. For a given supplier, this query lists out the parts along with the total loss incurred on the sales of that part. The cursor in the UDF iterates over each lineitem with the specified part, and computes the profit gained. If the profit is less than zero, i.e., it is a loss, then it is accumulated in the total_loss variable.

Loops result in a cycle in the control-flow graph of the UDF, making the task of algebraizing them challenging and, in some cases, impossible. Since queries involve disk IO, our main aim is to decorrelate queries inside a UDF with respect to the outer query block. Now, we describe techniques to decorrelate UDFs with cursor loops, and table valued UDFs. Later, we discuss how our approach could be extended to arbitrary while loops.

Example 5 UDF with a Loop

```java
create function totalloss(int pkey) returns int as
begin
  int total_loss = 0;
  int cost = getCost(pkey);
  declare c cursor for
    select price, qty, disc from lineitem
    where partkey:=pkey;
  open c;
  fetch next from c into @price, @qty, @disc;
  while @@FETCH_STATUS = 0
    int profit = (@price-@disc) - (cost * @qty);
    if (profit < 0)
      total_loss = total_loss - profit;
    fetch next from c into @price, @qty, @disc;
  close c; deallocate c;
  return total_loss;
end
```

Query: select partkey, totalloss(partkey)

```sql
from partsupp where suppkey = ?;
```

A. Algebraizing cursor loops:

The first step to building an expression for a cursor loop is to build an expression for the body of the loop. The body of a loop may contain imperative statements, query execution statements and nested loops, with arbitrary data dependences [8] between them. Such interstatement data dependences are captured by a data dependence graph (DDG) using static analysis of the code. The key difference between statements in a loop body and other statements which are not part of a loop is that statements in a loop may have cyclic data dependences [9], [8], i.e. loops may result in cycles in the DDG. For instance, consider the loop in Example 5. The value of variable total_loss, written in an iteration, is read in the subsequent iteration, resulting in a cyclic data dependence.

The parameterized expression for a loop with no cyclic data dependences is built as follows. Let $$E_{c}$$ be the expression for the query on which the cursor is defined, and let $$E_{b}$$ be the expression for the body of the loop. Expression $$E_{b}$$ is constructed using the technique described in Section IV. Then the expression for the loop is:

$$E_{l} = (S \times E_{c}) \times A^M E_{b}$$

The presence of cyclic dependences in a loop make it impossible to construct a set oriented algebraic expression for the loop in its given form. However, cyclic dependences are quite commonly encountered in loops in UDFs. We now describe how to compute the expression $$E_{b}$$ for the body of a loop with cyclic data dependences, using user defined aggregate functions.

Consider the subgraph of CFG corresponding to the body of a loop. Let the logical nodes in this subgraph be $$L = L_1, \ldots, L_k$$. Let $$L_i$$ be the first node in $$L$$ that is part of a cycle of data dependences. Then, the contribution of nodes $$L_i, \ldots, L_k$$ (referred to as $$L_c$$) can be captured as a user defined aggregate function, if the following conditions hold:

1) The initial values of all variables written in $$L_c$$ are statically determinable. This is because initial values for
Let $E_{in}$ be the expression at the point that precedes $L_i$. $E_{in}$ is constructed as described earlier (the fetch next statement is modeled as an assignment). Then the expression constructed for the body of loop $L$ is:

$$E_b = G_{f_c}((E_{in})$$

where $f_c$ is the auxiliary function created for nodes $L_c$. $f_c$ is a tuple-valued aggregate function with the signature:

$$\text{TUPLE}(c_1, \ldots, c_k) \rightarrow f_c(b_1, \ldots, b_m)$$

where (i) $c_1, \ldots, c_k$ are the variables that are live at the end of loop $L$, and (ii) $b_1, \ldots, b_m$ are the attributes that statements in $L_c$ use, but do not modify. The body of $f_c$ is constructed using the statements in $L_c$, and $f_c$ is initialized with the set of variables that are written to by statements in $L_c$. Nested loops are not considered in the above description, but can be handled similarly; we omit the details.

In the loop of Example 5, the cyclic dependence is present in the following logical node:

if (profit < 0)

The expression is computed up to this logical node as described earlier. The variable $total_loss$ is the only variable written to in this node, and its initial value can be statically determined to be 0. Therefore this logical node is expressed as a user defined aggregate function that accepts $profit$ as its parameter, and returns $total_loss$.

User defined aggregate functions should support a set of methods that are invoked at different stages during their evaluation [10]. In particular, they should support an initialization method where initial values are set, an accumulate or iterate method that accumulates individual input values, and a terminate method that returns the aggregate value. For Example 5, our technique results in a user defined aggregate function $aux-agg()$ shown in Example 6. Observe that the accumulate method contains the same code as the logical node with a dependence cycle.

The final expression constructed for the query in Example 5 is shown in Figure 8 where the function $aux-agg()$ is the user defined aggregate. Merging the expression tree with the outer query block and removing correlations is done as described in Section V and Section VI.

Note that in this approach, we conservatively move all the statements in $L_i$ into an aggregate function. In other words, all the statements in the loop that follow $L_i$ (the first statement that is a part of a dependence cycle) are considered for the user defined aggregate. However, there could be statements in $L_i$ that are not part of any dependence cycle. In such cases, it may be possible to reorder statements in the loop such that $L_c$ contains only statements that are part of a dependence cycle, or statements dependent on them. This optimization is an ongoing work, and details are beyond the scope of this paper.

### B. Algebraizing table valued UDFs

Table valued UDFs that build and return a temporary table are encountered very often in applications. Such UDFs typically look like the one shown in Example 7. This UDF creates a temporary table, iterates over a cursor and inserts values into a temporary table in every iteration before returning...
the table. Table valued UDFs can be represented algebraically using our technique if (i) the loop does not contain cyclic data dependences, (ii) there are no updates or deletes to the table valued attribute (only inserts are present), and (iii) the table valued attribute is not modified both before and after the loop.

The expression for the cursor loop is built as described earlier. The statement that inserts values into the temporary table is algebraized by using a projection on the attributes of the temporary table. In the UDF of Example 7, let $E_c$ be the expression for the query on which the cursor is defined. Let $(a_1, a_2, \ldots)$ be the attributes of the temporary table $tt$, and let $E_b$ be the expression for the code that computes the values $(\@v_1, \@v_2, \ldots)$ that are inserted into $tt$. Then, the expression for the UDF is:

$$(((S \ A^x \ E_c) \ A^M \ E_b) \ A^x \ \Pi_{v_1 \ as \ a_1,v_2 \ as \ a_2,\ldots}(S))$$

C. Discussion

The ideas presented in this section can be used to build an algebraic representation for scalar or table valued UDFs with cursor loops. This covers a large class of UDFs commonly encountered in practice. However, these ideas cannot handle loops with a dynamic iteration space, such as arbitrary while loops. Since queries involve disk I/O, our main goal is to decorrelate any queries inside loops. Techniques such as loop fission [9], [11] can be extended to isolate query execution statements into separate loops. The query execution statements can then be decorrelated, while the rest of the loops could remain as auxiliary UDFs, since they may have cyclic dependences that may make it impossible to construct set-oriented algebraic expressions. Handling loops with dynamic iteration spaces is ongoing work.

VIII. Related Work

Queries containing user-defined functions, such as the one shown in Example 1, can be thought of as nested queries with complex inner (sub-query) blocks. In the case of nested queries, the inner block is simply another SQL query with correlation variables used as its parameters. However, UDFs typically use a mix of imperative language constructs and SQL, and queries inside UDFs are embedded inside procedural code. Over the last three decades, there has been a lot of work on efficient evaluation of nested subqueries by decorrelating them. However, query decorrelation techniques proposed till date, such as [2], [12], [4], [5], [3], [6], cannot be used to decorrelate queries present in UDFs. The techniques presented in this paper enable decorrelation of UDF invocations.

Chaudhuri and Shim [13] consider optimization of queries containing user-defined predicates. Their work addresses the problem of choosing an optimal order for evaluating the predicates and joins in the query. Lieuwen and DeWitt [14], consider the problem of optimizing set iteration loops in database programming languages. Guravannavar [15] considers the problem of rewriting queries inside program loops to make use of parameter batching. Cheung et al.,[16] address the problem of automatically partitioning database application code where part of the code runs inside the database as stored procedure, with the goal of reducing latency. Similar to these works, our approach makes use of information about data dependencies and transformations such as loop fission and statement reordering originally proposed in the context of parallelizing compilers [11], [17], to enable decorrelation of queries inside UDFs.

IX. Implementation

Our techniques can be used with any database that supports UDFs and standard decorrelation transformations. The extensions and equivalence rules proposed by us can be integrated with the query optimizer to enable decorrelation of UDF invocations.

In order to measure the effectiveness of our techniques and transformation rules, we have implemented them as a query rewrite tool which can be used as a preprocessor for a database system. The tool accepts a database schema, an SQL query, and definitions of UDFs used by the query, written in the syntax of a commercial database system (SYS1), as its inputs. It produces as output a rewritten SQL query along with definitions of auxiliary functions, if any, used by the rewritten query. The rewritten SQL query is then executed on the database system, which performs cost-based optimization on the query.

The structure of the rewrite tool is shown in Figure 9. After parsing, a tree structured intermediate form of the query and the referenced UDFs is constructed. If the UDF contains loops, the loop fission module may be used to perform the necessary transformations while the tree is built. This tree makes use of Apply operators with extensions as described in this paper. The aux function builder is invoked as required to generate auxiliary functions. Transformation rules that remove the Apply operators are then applied to the intermediate tree form. If the tool is unable to remove all the Apply operators, it does not transform the query. Finally, the output phase generates a SQL query and auxiliary functions from the transformed intermediate representation.

The implementation for handling loops in UDFs is in progress. In particular, some databases (including SYS1) do not allow user-defined aggregates to be written in procedural SQL. Automating the creation of non-SQL functions from SQL UDFs is a work in progress. Hence in the experiments which deal with UDFs containing loops, we have manually performed the transformations in accordance with our techniques.

X. Experimental Results

We have designed and conducted experiments to assess (a) the applicability of proposed rewrite techniques to real-world UDFs, and (b) the performance benefits due to the rewrite on modern commercial database systems. To the best of our
Example 8 UDF for Experiment 1

```sql
create function discount(float amt, int ckey) returns float as
begin
    int custcat; float catdisct, totaldiscount;
    select category into :custcat
    from customer where customerkey = :ckey;
    select frac_discount into :catdisct
    from categorydiscount where category = :custcat;
    totaldiscount = catdisct * amt;
    return totaldiscount;
end
```

Query: select orderkey, discount(totalprice, custkey) from order;

knowledge, there is no benchmark for SQL where queries make extensive use of UDFs. To assess the applicability of our rewrite techniques, we constructed and borrowed UDFs from real-world applications. These UDFs make use of various constructs offered by a typical imperative language. The program logic in most of these UDFs is influenced by functions and procedures found in real-world applications, and changes were made primarily for running them against the TPC-H dataset.

We have performed our experiments on two widely used commercial database systems - SYS1 and SYS2. Our tool was used to generate the decorrelated queries for SYS1, and they were manually translated to the syntax of SYS2 in order to run the experiments. The database servers were run on Intel Core i5 3.3 GHz machines with 4 GB of RAM. The queries were run locally on the TPC-H 10 GB dataset with a few augmented attributes to suit our examples. The tables customer and orders had 1.5 million and 15 million records respectively, with default indices on primary and foreign keys.

Experiment 1: As the first experiment, we consider a UDF which computes the discount for a customer based on the category of the customer. The UDF and the query are shown in Example 8. The UDF has no branching or loops, and has a sequence of straight line code. This UDF executes two scalar SQL queries and an arithmetic operation in order to compute the discount value. After applying our technique, we get the following decorrelated form for the query of Example 8:

```sql
select o_orderkey,(frac_discount*o_totalprice) as totaldiscount
from orders o, customer c, categorydiscount cd
```

where o_custkey=c_custkey and c_nationkey=custcategory

Figure 10 shows the results on SYS1 with the number of UDF invocations on the x-axis and the time taken on the y-axis. We vary the number of UDF invocations by using a top clause. We can see that for smaller number of invocations, both the original and the rewritten query perform similarly. The optimizer performed an iterative invocation of the UDF for the original query; it chose a plan with nested loop joins for the rewritten query. However, as the number of invocations increase, the time taken by the original query steadily increases. This is because the optimizer does not have alternative plans to choose, and uses the same iterative plan.

In contrast, the time taken by the rewritten query remains very low even with a larger sizes as the optimizer chose other plans with hash join. This shows how our transformations enable the optimizer to choose better plans. Similar patterns were observed on SYS2 though the actual numbers vary. For instance, at 1 million invocations, the time taken by the original query and the rewritten were respectively 6 minutes and 14 seconds. At 10 million invocations, the original query took 16 minutes whereas the rewritten one ran in 2 minutes.

Experiment 2: We consider the query and UDF shown in Example 1 of Section I, with its rewritten form in Example 2. Recall that this UDF has assignment statements, branching statements and a scalar SQL query.
We vary (by appending a where clause) the number of customers, and hence the number of UDF invocations, and report the time taken by the original and transformed queries. The results on SYS1 are shown in Figure 11 with the UDF invocation count on the x-axis and the time taken (in log scale) on the y-axis.

The observations here are similar to what we observed in Experiment 1. Observe that up to 1K invocations, both the original and transformed queries perform similarly. For the original query, the optimizer chose a plan which iteratively invokes the UDF for each tuple in the customer table. In the case of the transformed query, the optimizer chose a nested loops join, thus resulting in similar performance. As the number of customers increase, the original query plan remains the same, and hence performance degrades.

For the rewritten query, the time taken actually reduces between 1K and 10K before starting to rise very gradually for invocations beyond 10K. This drop is due to the fact that up to 1K, the chosen plan had two nested loop join operations. Between 1K to 5K, one of them switches to a hash join; between 5K to 10K, the second one also switches to a hash join. At 10 million customers, the original query took more than 3 hours whereas the rewritten query ran in less than a minute. Similar trends were recorded on SYS2 as well, where the rewritten query took about 9 minutes while the original query ran for almost 24 hours.

**Experiment 3:** We consider a UDF with a loop borrowed from [15]. The UDF computes the number of parts in a given category and all its parent categories. The parts table had 2 million rows and there were 1000 categories. Since our tool currently does not handle UDFs with loops, we have manually applied the transformation rules presented in this paper. Similar to Experiment 2, we vary the number of UDF invocations by appending a where clause on the categories table, and record the time taken. The results of this experiment on SYS1 are shown in Figure 12, where the x-axis indicates the number of UDF invocations and y-axis shows the time taken.

Similar to earlier experiments, the time taken by the original query increases as the number of invocations increases. We observe that for smaller number of invocations, the transformed query actually performs a bit worse than the original query. In fact, as the graph shows, the time taken by the rewritten query is a constant (at 5 seconds). This is due to the fact that the scan on the parts table dominates the query execution time, and the selection operation does not reduce this.

Our technique of decorrelating UDFs is designed to be part of a cost-based optimizer. If an optimizer incorporates these techniques, it can choose the better of the two plans for smaller number of invocations, since iterative invocation remains as an alternative. Since our current implementation is an external tool, this option is not available to the optimizer. At larger number of invocations, however, the rewritten form turns out to be significantly faster than the original query.

**XI. Conclusion and Future Work**

While there are many reasons to use complex UDFs in queries, they have not been widely used since query optimizers tend to produce inefficient correlated execution plans for queries with such UDFs. Our work is the first to show how to decorrelate UDF invocation, and can be used as part of cost-based optimization, or as a query rewrite technique. Our performance results show up to orders of magnitude performance improvement on two commercial database systems. Thus our technique could play a key role in increasing the usage of complex UDFs. More generally, our work is the first to provide an algebraic representation for UDFs, which could have uses beyond decorrelation in optimization of queries invoking UDFs.

As part of future work, we first plan to complete our implementation to handle UDFs with cursor loops. Our techniques need to be extended to decorrelate UDFs with arbitrary while loops. We also currently do not handle updates and deletes to table valued attributes. We also plan to extend our techniques to UDFs written in languages other than SQL. The current work focuses on UDFs which are pure functions with no side effects. Extending these ideas to optimize stored procedures is another interesting area for future work.

**Acknowledgements:** We thank Amey Karkare for his comments, and Subhro Bhattacharyya for help with experiments.

**References**


