A Framework for Inferring Properties of User-Defined Functions

Xinyu Liu, Joy Arulraj, Alessandro Orso
Georgia Institute of Technology
Atlanta, Georgia, USA
liuxy@gatech.edu|arulraj@gatech.edu|orso@cc.gatech.edu

ABSTRACT

User-defined functions (UDFs) are widely used to enhance the capabilities of DBMSs. However, using UDFs comes with a significant performance penalty because DBMSs treat UDFs as black boxes, which hinders their ability to optimize queries that invoke such UDFs. To mitigate this problem, in this paper we present LAMBDA, a technique and framework for improving DBMSs' performance in the presence of UDFs. The core idea of LAMBDA is to statically infer properties of UDFs that facilitate UDF processing. Taking one such property as an example, if DBMSs know that a UDF is pure, that is it returns the same result given the same arguments, they can leverage a cache to avoid repetitive UDF invocations that have the same call arguments.

We reframe the problem of analyzing UDF properties as a data flow problem. We tackle the data flow problem by building LAMBDA on top of an extensible abstract interpretation framework and developing an analysis model that is tailored for UDFs. Currently, LAMBDA supports inferring four properties from UDFs that are widely used across DBMSs. We evaluate LAMBDA on a benchmark that is derived from production query workloads and UDFs. Our evaluation results show that (1) LAMBDA conservatively and efficiently infers the considered UDF properties, and (2) inferring such properties improves UDF performance, with a time reduction ranging from 10% to 99%. In addition, when applied to 20 production UDFs, LAMBDA caught five instances in which developers provided incorrect UDF property annotations. We qualitatively compare LAMBDA against Froid, a state-of-the-art framework for improving UDF performance, and explain how LAMBDA can optimize UDFs that are not supported by Froid.

KEYWORDS

UDF properties, DBMSs, static analysis

1 INTRODUCTION

Database applications often contain user-defined functions (UDFs) written in imperative languages such as Java. For instance, tens of millions of UDFs are in use in the AzureSQL database-as-a-service (DBaaS) platform with billions of daily invocations [25]. This is because UDFs offer many advantages over standard SQL queries: (1) allow code reuse across queries thereby improving code modularity; (2) allow expression of complex business rules that cannot be expressed easily using SQL; (3) allow a combination of SQL and imperative code to improve readability and maintainability.

Challenges. While UDFs improve application developer experience, they may also lead to slower queries. In fact, practitioners have extensively documented the performance overhead of UDFs [12, 13, 25]. The root cause of the runtime performance overhead imposed by UDFs is that the query optimizer of the database management system (DBMS) typically treats UDFs as a black box. Therefore, the DBMS cannot effectively optimize queries that invoke UDFs, leading to less efficient query plans and slower query execution. n

Existing Techniques. There are currently two main kinds of techniques for tackling the problem of inefficient UDF processing.

The first type of techniques focuses on transforming UDFs written in procedural extensions of SQL (*e.g.*, T-SQL) to semantically equivalent SQL subqueries [15, 18, 25, 28]. By eliminating UDFs, this UDF translation technique enables the DBMS's optimizer to apply more query optimization rules on UDF-invoking queries. Froid [25] is a popular example of this kind of techniques and has been integrated into Microsoft's SQLServer. However, UDF translation techniques have two key limitations. First, to guarantee that the translation does not break the original query's semantics, they are limited to a subset of imperative language constructs. For instance, Froid does not support UDFs with more than one return statement. Second, these techniques may transform UDFs into convoluted subqueries that lead to even worse query performance [4, 25]. To mitigate this problem, Froid imposes a constraint on the size of the query to which it applies UDF transformations, thereby limiting the space of queries that can be optimized.

The second type of approaches consists of manually annotating the properties of UDFs. In this case, the DBMS exports an annotation interface that allows users to inform the query optimizer about certain properties of each UDF. Once the DBMS is aware that a UDF satisfies a certain property, it can evaluate queries containing these UDFs more efficiently, by leveraging optimizations that are otherwise impossible to apply. For instance, if the DBMS knows the UDF always returns the same result given the same arguments, it can cache the UDF's results, which eliminates redundant UDF calls. While DBMSs often support such UDF property annotation interfaces, they rely on users to provide correct annotations. Furthermore, providing accurate UDF annotations is crucial because incorrect annotations may lead to incorrect query results or sub-optimal query performance. However, it is challenging for DBMS users (*i.e.*, application developers) to provide correct UDF annotations, given the complexity of manually analyzing code. This approach is therefore error-prone and unreliable. In fact, in a case

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639147

study we performed on a set of production UDFs, we found that only 50 percent of the UDFs had correct annotations (§5).

Our Approach. Given the limitations of existing techniques and the potential performance benefits of leveraging UDF properties, we present an automated technique for generating UDF annotations based on program analysis that relies on two key insights. First, UDF property annotations can help DBMSs process UDFs that translation-based techniques cannot handle due to their inherent limitations (§2.2). Second, the problem of verifying UDF properties can be formulated as a dataflow analysis problem. As we design the analysis in a way that overapproximates UDF behaviors, our technique conservatively catches all instances of property violations. That is, if our technique may miss an annotation, that may result in a missed query optimization, but will never generate an incorrect one, thus avoiding spurious optimization and ultimately incorrect results.

We combine these insights in a technique named LAMBDA, which takes as input a UDF and verifies whether the UDF satisfies a set of relevant properties. Conceptually, LAMBDA is similar to traditional compilation techniques that compute program properties to generate more optimized programs [24, 27]. Currently, LAMBDA supports four UDF properties that are commonly used across DBMSs [5–7] (see §3). To automatically and soundly determine these properties, LAMBDA performs dataflow analysis using an abstract interpretation framework [1] that ensures an overapproximation of program behaviors over the property of interest.

We evaluated LAMBDA on a UDF benchmark that is derived from production query workloads and real-world UDFs. Our results are promising, as LAMBDA was able to: (1) improve query performance with a time reduction ranging from 10% to 99%, and (2) infer these properties from 30 production UDFs conservatively and efficiently, with zero false negatives (see Figure 6) and an average analysis time of 34.9 milliseconds for each UDF. LAMBDA also caught five instances wherein developers incorrectly annotated the UDFs, all of which have been reported and acknowledged. We compared LAMBDA against Froid, a technique based on transforming UDFs into equivalent SQL subqueries. Our results show that LAMBDA was able to improve query performance in cases that Froid could not handle. We defer the integration of LAMBDA into a data-processing platform to future work. This paper makes the following contributions:

- An technique for automatically inferring UDF properties based on data-flow analysis that can considerably improve the performance of DBMSs in processing UDF-invoking queries.
- A publicly available implementation of the technique [9].
- A publicly available UDF benchmark [9].
- An evaluation of our technique that shows that (1) LAMBDA can infer the UDF properties it supports soundly and efficiently, (2) the inferred properties can lead to considerable performance improvements, and (3) LAMBDA can process queries that alternative approaches cannot handle.

2 BACKGROUND

This section presents relevant background information, a motivating example, a case study about UDF annotations, and an overview of data flow analysis.

Listing 1: An example Java UDF that challenges existing approaches.

```java
public Integer processDate(String strBeginDate, Integer n) {
  Date dtBegin = new Date();
  try {dtBegin = (new
       SimpleDateFormat("yyyyMMdd")).parse(strBeginDate);
  } catch (ParseException e1) {e1.printStackTrace();}
  Calendar cld = Calendar.getInstance();
  cld.setTime(dtBegin);
  int startMonth = cld.get(Calendar.MONTH);
  try {cld.setTime(dtBegin);
    cld.set(Calendar.MONTH, startMonth + n);
    Date dtNew = cld.getTime();
  } catch (NumberFormatException e) {e.printStackTrace();}
  String query = "SELECT d_date FROM store_sales, date_dim
       WHERE ss_sold_date_sk = d_date_sk ORDER BY d_date";
  ResultSet rs = query.executeQuery();
  Date queryresult = new Date();
  queryresult = rs.getDate("d_date");
  return dtNew.compareTo(queryresult); }
```

2.1 Java UDFs

Imperative languages offer several benefits over the declarative SQL, such as better maintainability, readability, and supporting expression of complex business logic. For this reason, many DBMSs (e.g., PostgreSQL, MySQL, and Apache Hive) allow users to write UDFs in traditional imperative languages, such as Java. We notice that there are tens of millions of Java UDFs in use today in a proprietary cloud database system, with millions of daily invocations. These UDFs cover a wide variety of functionalities, such as processing a many different data types (*e.g.*, dates, strings, and url) and implementing complex business logic. Another empirical observation is that each UDF is written within an individual Java class and developed as a member function. When a Java UDF is invoked, the class instance that contains this UDF is first initialized, and then the UDF is evaluated iteratively for each qualifying tuple.

2.1.1 Optimization and Execution of Java UDFs. Since DBMSs currently treat Java UDFs as black boxes, queries that contain UDFs are under-optimized and thus processed inefficiently (as demonstrated in Figure 1). This is because many query optimization techniques can only be applied on UDF-invoking queries if the referenced UDFs are guaranteed to satisfy certain properties [5–7]. While DBMSs currently allow developers to provide property annotations, manually analyzing UDFs is time-consuming and error-prone. As a result, an automated analysis approach such as LAMBDA is a valuable technique that can improve the existing practices of UDF processing. Another notable fact is that Java UDFs are evaluated within a Java runtime instead of a query execution engine. This means that executing UDFs incurs an unavoidable and expensive performance cost due to repeated context switching between the Java runtime and the query execution engine [14]. LAMBDA helps DBMSs reduce such costs by enabling optimizations that save unnecessary UDF invocations.

2.2 Motivating Example

We present a Java UDF to demonstrate the challenges associated with existing approaches and the potential benefits of leveraging UDF properties. Listing 1 shows a UDF derived from production UDFs. This UDF first modifies a given date by incrementing its month, and then compares the date with the result of a query. The

```
SELECT processDate('19000101',5)
WHERE processDate('19000101',5) IS NOT NULL;
```

(a) Original query Q1, execution time = 5 s

```
SELECT processDate('19000101',5);
```

(b) Optimized query Q2, execution time = 2.4 s

Figure 1: An example optimization enabled by a UDF property.

UDF can be broken into five main steps. ❶ It parses the input string into a date object *dtBegin* (line 2 – 4). ❷ It initializes a calendar object *cld* using *dtBegin* and updates it by incrementing its month (line 5 – 9). ❸ It initializes a new date object *dtNew* using *cld* (line 10). ❹ It sends a query to the connected database and stores the result in *queryResult* (line 12 – 15). ❺ It returns the comparison result between *dtNew* and *queryResult* (line 16). Notably, for the first step, the UDF leverages the exception construct to handle cases in which the string parsing fails.

Why this case is challenging for existing techniques. Translation-based approaches cannot transform this UDF into a SQL query because SQL does not support exceptions. Moreover, this UDF also makes it difficult for UDF users to provide correct annotations. Case in point, it corresponds to a confirmed UDF annotation bug that our technique discovered: the UDF was incorrectly annotated as pure (see §3 for a rigorous definition), a property that guarantees that this UDF always generates the same output given the same inputs. However, this UDF is in fact impure because it can generate different outputs for the same inputs. In particular, when a parse exception occurs, the UDF will calculate its output based on the system time (line 2), which can lead to different outputs for the same inputs. Incorrect property annotations are harmful because they can often lead to incorrect query results.

How UDF properties improve query performance. Figure 1 shows how DBMSs can leverage UDF properties to speed up UDF processing. Consider Q1 (Figure 1a), which asks the DBMS to return the result of the example UDF only if the result is not NULL. By manually analyzing this UDF, we can conclude that it never returns NULL (see how LAMBDA automatically analyzes it in §4). By knowing this UDF property, DBMSs can remove the filter predicate within Q1 while preserving the query semantics. Q2 is the result of applying this optimization, which eliminates unnecessary calls to the UDF within the filter predicate. Consequently, Q2 runs 2× faster than Q1 on the same database in PostgreSQL (v14.3).

2.3 Case Study

We present a case study based on a production system to demonstrate the potential usefulness of our approach in practice. We conducted the case study about UDF annotations on a general-purpose, fully managed, multi-tenancy data processing platform for large-scale data warehousing [10]. For the platform's daily production workload, hundreds of worker nodes are invoked to query massive datasets (size is up to hundreds of TB). We examined a random sample of 20 Java UDFs that cover a wide variety of functionalities, such as dates, strings, and URL processing, mathematic computation, and implementing complex business logic. We focus on the Purity property for this case study and defer its formal definition to §3. We have 2 evaluation goals: ❶ whether developers provide correct property annotations for production UDFs and ❷ whether UDF properties improve the system performance. We focus on the

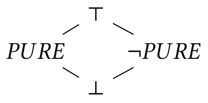


Figure 2: Dataflow facts for Purity.

Purity property for this case study and defer its formal definition to §3.

Property annotations in production UDFs. We first manually analyzed the sampled UDFs to determine whether they satisfied the four properties considered and used the result of the analysis as the ground truth. Then, we compared existing annotations (provided by UDF users) on these UDFs against the ground truth. Interestingly, we found that UDF annotations are often incorrect. Among 20 sampled UDFs, 10 had incorrect annotations: 5 stated that a property was satisfied when it was not, and 5 stated that a property was not satisfied when it was. This result demonstrates manually analyzing UDFs is error-prone and an automated and sound approach is necessary and highly desirable.

Performance improvement of leveraging UDF properties. We next examined the platform's performance improvement after leveraging the UDF property. We first collected 615 production queries that invoke the sampled UDFs that satisfy the Purity property. Then, we ran these queries two separate times, with and without providing the Purity annotations to the platform, and compared the total of their query execution time, machine instance time and cpu cost. The result shows that leveraging Purity annotation improves the system performance from all measurements, with 2.2%, 4.8%, and 4.9% improvements, respectively. More efficient query execution plans and computation results reuse are two main contributing factors behind these improvements. Given the expensive cost of performing large-scale data processing tasks, this improvement can lead to significant reductions in computational resources consumption and monetary spending. Moreover, the platform administrators confirm that the results are generalizable to other properties, which suggests that providing additional properties' annotations to the system can further increase the magnitude of the performance improvement.

2.4 Dataflow Analysis

Dataflow analysis (DFA) is a static analysis technique that reasons about program properties. DFA has many applications, such as compiler optimization and software vulnerability detection. DFA operates on a program's control-flow graph (CFG), with the goal of computing information that is guaranteed to hold at each program point (*i.e.*, each node of the CFG) on all executions.

To achieve this, DFA builds upon three fundamental elements: **dataflow facts**, **flow functions**, and **join operator**.

Dataflow Facts. They represent program information on which the analysis focuses and can be formally defined using a lattice. The goal, in defining dataflow facts, is to abstract away concrete program information while keeping necessary information for the analysis.

Figure 2 is an example lattice that defines the dataflow facts for modeling Purity information that each program variable holds. The explanation for this lattice is as follows: for a variable at a given program point, when the program inputs are the same, whether this variable always has the same value (maps to the *PURE* from the lattice) or not (maps to the $\neg PURE$ from the lattice). For instance, if a variable holds a constant value at a program point, it maps to a *PURE* dataflow fact because constant values are always pure. In addition, $\bot$ and $\top$ are two standard dataflow facts to complete the lattice: $\bot$ models the program point that has not been analyzed; $\top$ models cases wherein more precise dataflow facts are unavailable. For instance, when different execution paths make it possible for a variable to hold different dataflow facts (either *PURE* or $\neg PURE$) at the same program point, we assign $\top$ to this variable.

Flow Functions. They model the effect of instruction on dataflow facts. Considering the dataflow analysis for Purity, for instance, the flow function for an assignment statement is to propagate the dataflow fact from the right to the left.

Join Operator. It defines how to compute dataflow facts at a join point (a node with more than one predecessor). It usually performs a union operation over the facts from each predecessor.

Iterative Dataflow Analysis Framework. Dataflow analysis problems usually are solved using the standard iterative algorithm [20]. The algorithm uses flow functions and the join operator to propagate dataflow facts over the CFG of the program and performs iteration until the dataflow facts associated with each CFG node converges to a fixed point [20].

3 PROBLEM FORMULATION

Now we formalize the problem of analyzing UDF properties. In particular, we present formal definitions of four UDF properties: Purity, Non-null, Returns Null On Null, and Parallel Safe. We define four UDF properties that are widely adopted in existing DBMSs. If UDFs satisfy any of these properties, DBMSs can apply an additional set of optimization techniques on queries that invoke these UDFs, which leads to a much quicker query response time. By default, DBMSs assume UDFs do not satisfy any of these properties.

Property 1. Purity: A UDF satisfies Purity if it guarantees to return the same result given the same arguments and does not have side-effects (*e.g.*, modifying the database state).

Relevant optimizations. This UDF property enables both single-query and multi-query optimization techniques, by allowing DBMSs to reuse UDF results. For single query optimizations, a pure UDF allows the DBMS to save redundant calls with the same argument. In particular, DBMSs can either optimize multiple calls of the function to a single call or maintain a cache to store previous UDF evaluation results; both techniques help reduce the UDF evaluation time and thus improve the query performance. In addition, a pure UDF allows the DBMS to use multi-query optimization techniques, such as common subexpression elimination [2] and materialized views [3], which often lead to more efficient query execution plans.

Property 2. Non-null: A UDF satisfies Non-null if it guarantees to return a non-NULL value.

Relevant optimizations. This property facilitates both UDF optimization and execution. If a UDF is Non-null, DBMSs can remove operations that compare the UDF result with NULL.

Property 3. Returns Null On Null: A UDF satisfies Returns Null On Null if the function always returns null whenever any of its arguments is null.

Relevant optimizations. This property facilitates UDF evaluation, as DBMSs do not execute such functions when they have null arguments and automatically assume their result is NULL. By reducing unnecessary UDF calls, this optimization reduces the number of context switches between the JVM and the SQL query execution engine, which indirectly reduces the overall cost of UDF execution.

Property 4. Parallel Safe: A UDF satisfies Parallel Safe if the function can be evaluated in the parallel mode.

Relevant optimizations. This property allows DBMSs to launch more than one worker to execute the calling query concurrently. Different DBMS implementations have different requirements for a UDF to be Parallel Safe. Taking PostgreSQL as an example, a Parallel Safe UDF cannot perform certain operations that cannot be synchronized across parallel workers, such as modifying database table contents.

4 KEY CONCEPTUAL INSIGHTS

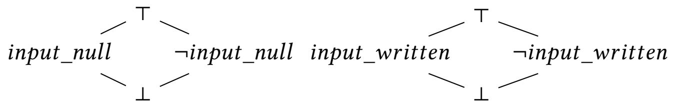
We now discuss our key insights for analyzing UDF properties defined in §3. We show that the verification of UDF properties can be represented as a general dataflow problem that detects possible violations of UDF property specifications (a UDF satisfies a UDF property if its behaviors do not violate the property specification). The dataflow problem is parameterized by three main components: dataflow facts, flow functions, and join operators. For ease of presentation, we first present commonalities and then differences among dataflow problems for distinct UDF properties.

4.1 Commonalities

DFAs for UDF properties share the same iterative dataflow analysis framework [21]. Algorithm 1 shows its algorithm. The framework takes as input a UDF, its CFG, and a property P, and computes as output whether the UDF satisfies P. To achieve this, the framework associates each CFG node with an analysis state, σ, that tracks relevant information for the property P. In particular, σ maintains a mapping between each CFG node and dataflow facts l that are relevant to P. The framework performs dataflow analysis in four main steps. ❶ For all CFG nodes, the framework initializes their input and output analysis state to $\bot$ (i.e., an analysis state that has not yet been processed), and adds each node to a worklist (line 3). ❷ The framework performs the following iteratively, until the worklist is empty (implicitly indicates that the analysis state for each node has reached a fixed point): (1) the framework takes a node off from the worklist, and computes its incoming analysis state by calling the processJoin procedure on its predecessors; (2) the framework computes the new outcoming analysis state for this node by invoking the processBlockFlow procedure that takes as input the node and its incoming analysis state; (3) if the newly computed outcoming analysis state is different from the previous one, the framework performs 2 things: updating this node's outcoming analysis state and adding its successors to the worklist; otherwise, it does nothing. ❸ After the worklist becomes empty, the framework invokes processReturnBlocks. Within this procedure, the framework calls the processJoin procedure on the CFG's return blocks to conservatively compute the return state of the UDF. Then, it invokes the checkFinalStateAgainstProperty procedure to determine whether the return state satisfies the UDF property P.

Algorithm 1: Iterative Dataflow Analysis Algorithm.

```
Input  : a UDF and its CFG; a property P
Output : whether the UDF satisfies P
worklist = [ ] ;
input_state[n] = output_state[n] = { };
for Node n ∈ CFG do
    input_state[n] = output_state[n] = ⊥ ;
    worklist.ADD(n) ;
while worklist! = ∅ do
    Node n = worklist.POP() ;
    for Node k ∈ PREDS(n) do
        input_state[n] =
          PROCESSJOIN(input_state[n], output_state[k]);
    new_output_state = PROCESSBLOCKFLOW(n, input_state[n])
    if new_output_state != output_state[n] then
        output_state[n] = new_output_state ;
        for Node j ∈ SUCCS(n) do
            worklist.ADD(j) ;
return PROCESSRETURNBLOCKS(output_state)
Procedure PROCESSBLOCKFLOW(Node n, input_state[n])
    output_state = input_state[n] ;
    for Instruction i ∈ GETINSTRUCTIONS(n) do
        output_state = FLOWFUNCTION(i, output_state) ;
    return output_state ;
Procedure PROCESSRETURNBLOCKS(output_state)
    final_state = ⊥ ;
    for Node n ∈ GETRETURNBLOCKS(CFG) do
        final_state = PROCESSJOIN(final_state, output_state[n]) ;
    return CHECKFINALSTATEAGAINSTPROPERTY(final_state, P) ;
```



(a) inputNull lattice **(b) inputWritten lattice**

Figure 3: Dataflow facts for the Returns Null on Null property.

4.2 Differences

DFAs for UDF properties differ in their answers to four key design questions for the framework: **1.** What program information to track (§4.2.1)? **2.** What operations impact the tracked program information (§4.2.2)? **3.** How to combine incoming program information for nodes that have more than 1 predecessor (§4.2.3)? **4.** How to check whether the UDF satisfies the property using the computed analysis states (§4.2.4)?

4.2.1 Dataflow Fact and Analysis State. In order to efficiently analyze different UDF properties, DFAs need to design different abstractions of program information. These abstractions can be described using domains of dataflow facts and formally represented as dataflow lattices. With a given domain of dataflow facts, a DFA maintains an analysis state, σ, to track information that each program component (e.g., either a program variable or a CFG node) holds. Table 1 summarizes the domains of dataflow facts and analysis states for analyzing different UDF properties.

```
1     public Integer processDate(String strBegDate, Integer n) {
2       if (strBegDate == null || n == null) {
3         return null;...}
```

Figure 4: An example code that processes NULL input.

Purity property requires a domain of dataflow facts that specify whether a program variable holds a pure value (*i.e.*, always evaluates to the same value given the same program inputs) at a given program point. This domain can be formally represented using a 2-element lattice shown in Figure 2. The analysis state maintains a mapping between each program variable and a value from the Purity lattice: at a given program point, a variable that maps to a $PURE$ value means it holds pure value, and a $\neg PURE$ value means the variable holds impure value.

Non-null property requires a domain of dataflow facts that specify whether a program variable holds a value that is not NULL, which can be formally represented using a 2-element (*i.e.*, $NULL$ and $\neg NULL$) lattice that has the same form as the Purity lattice. The associated analysis state maintains the mapping between each program variable and a value from the Non-null lattice: at a given program point, a variable that maps to a $NULL$ and a $\neg NULL$ value, means it is always NULL and not NULL, respectively.

Returns null on null property requires a domain of dataflow facts that specify a set of input parameters that guarantee to be NULL at each program point; we design the analysis domain by leveraging a key insight that an input parameter guarantees to be NULL if it succeeds a true branch that checks its equality against the value NULL and has never been overwritten. This insight is inspired by our empirical observations of how UDFs handle NULL inputs. Figure 4 demonstrates the programming pattern for handling NULL inputs. UDFs start with checking input parameters' equality against the NULL value (line 2). If these checks succeed, it guarantees that, within their successors, these input parameters are always NULL until they are assigned to a different value. As shown in our example code, the CFG node that contains line 3 is a successor to 2 distinct checks for inputs $strBegDate$ and n, and takes the true branch for both checks. Consequently, at the start of this CFG node, both $strBegDate$ and n must be NULL. Then, because it returns the NULL value as the output, we know that this UDF satisfies the Returns Null On Null property.

Based on these observations and insights, we design two domains of dataflow facts for analyzing this property, which can be formally represented using the *inputNull* and *inputWritten* lattice, respectively. The associated analysis state tracks two kinds of information: ❶ a set of input parameters that may have been written (using the *inputWritten* lattice); ❷ a set of parameters that must be NULL since they succeed successful equality checks against the NULl value (using the *inputNull* lattice). For instance, consider the parameter n in our example, at the start of the CFG node that contains line 3, it has $\neg input\_written$ and $input\_null$ as its dataflow facts. This is because the CFG node succeeds a successful equality check between n and NULL, and the variable is not rewritten.

Parallel safe property requires a domain of dataflow facts that specify whether a given CFG node contains operations that violate the parallel safe policy, which can be formally represented using a 2-element (*i.e.*, $safe$ and $\neg safe$) lattice that has the same form as the Purity lattice. The associated analysis state maintains the mapping between each CFG node and a fact value from this

Table 1: Dataflow fact domains and analysis states for UDF properties.

Property P	Within a given CFG node n, a dataflow fact l describes whether	Analysis state σ_n
Purity	a variable v_i holds a pure value	$\{\langle v_i, l\_pure\rangle, \ldots\}$
Non-null	a variable v_i holds a value that is not NULL	$\{\langle v_i, l\_nonnull\rangle, \ldots\}$
Returns Null On Null	an input variable v_i is never rewritten and succeeds a successful equality check against the NULL value	$\{\langle v_i, l\_null, l\_written\rangle, \ldots\}$
Parallel Safe	the node is free of operations that violate the parallel safe policy	$\{l\_safe\}$

Table 2: Example statements that may violate UDF properties.

Property P	Statements that may violate the property
Purity	Date dtBegin = new Date();
Non-null	return null;
Returns Null On Null	return Integer.valueOf(1);
Parallel Safe	String query = "INSERT ...";

lattice: $safe$ and $\neg safe$ means the node contains 0 and at least one operation that violates the parallel safe policy, respectively.

4.2.2 Flow Functions. At a high level, DFA focuses on analyzing program statements that may directly or indirectly violate the property definition. Table 2 demonstrates example statements that may violate each UDF property. Technically, for each property, DFA leverages a distinct set of flow functions that model the impact of program statements on associated analysis states. These flow functions are determined by the *property policy*, which enumerates the effect of operations (by the operator and its operands) and method invocations (by the method name and its parameters) on dataflow facts that are associated with each UDF property. Due to the limited space, we only present the *property policy* and a set of example *flow functions* for verifying the Purity property.

Property Policy for Purity. It contains a set of rules that can be expressed as a triple $\langle \mathcal{N}, \mathcal{C}, \mathcal{E}\rangle$. $\mathcal{N}$ specifies the name of the operator or the full name of the method. $\mathcal{C}$ specifies the calling context, such as whether the operator/method involves an impure operand/argument. $\mathcal{E}$ specifies the effect of the operator/method on the existing dataflow facts: *RAISE* and *KILL* means it generates a $\neg$*PURE* and a *PURE* dataflow fact, respectively. The following is a list of example rules for the Purity property:

Listing 2: an example list of purity rules.

```
(Multiplication, sizeOf(impure operands) > 0, RAISE)
(Date.setTime(), sizeOf(impure arguments) == 0, KILL)
```

Flow Functions for Purity Property. Flow functions are designed based on property rules. Consider the *Mutiplication* rule shown above as an example. Assume the analysis state before and after executing a multiplication statement is σ_{before} and σ_{after}, respectively. The flow function for the *Mutiplication rule* has 3 main steps: First, it assigns σ_{before} to σ_{after}. Second, it queries σ_{before} in order to check whether any operand maps to $\neg$*PURE* or $\top$ (defined by the calling context $\mathcal{C}$ of the rule). Third, if this is indeed the case, it updates σ_{after} by mapping the left-hand variable of the statement to $\neg$*PURE* (defined by the calling effect $\mathcal{E}$ of the rule); otherwise, it preserves σ_{after}.

4.2.3 Join Operator. Because a CFG node may have multiple predecessors, in order to compute its analysis state, we need the join operator that defines how to merge multiple incoming analysis states. For all properties that we consider, the join operator merges the incoming states by performing a union-like operation over each component's incoming dataflow values. Consider a state σ_{now} that has two incoming states, wherein the same relevant component *element* holds two different dataflow values, l_a and l_b. We compute the starting dataflow value for *element* at σ_{now} by performing a union operation over incoming dataflow values for *element*, which results in $\{l_a, l_b\}$. That is, at the beginning of σ_{now}, *element* may hold either l_a or l_b. For the Returns Null On Null property, the join operator includes an additional procedure. For a given CFG node, we iterate through its predecessors: if a predecessor ends with a successful equality check of an input variable against the NULL value, and the input variable has never been modified, we update the analysis state by assigning *input_null* to the input variable.

4.2.4 Analysis States and the UDF Property. Once a fixed point has been reached for all analysis states, the DFA performs a 2-step procedure to determine whether the UDF satisfies the property, by first computing the final analysis state and then checking whether the state satisfies the property.

Compute Final Analysis State. For all properties except the Returns Null On Null property, we compute the final analysis state by using the join operator over the analysis states for CFG nodes that have a return statement. To compute the final analysis state for the Returns Null On Null property, we apply the join operator over CFG nodes that return the NULL value. If none of the nodes returns the NULL, we conclude that the UDF does not satisfy the Returns Null On Null property.

Check Final Analysis State against the Property. For the Purity and the Non-null properties, the final analysis state includes a mapping between variables and dataflow values that indicate whether variables are pure and not null, respectively. If variables used as the return value are pure and not null, the DFA concludes that the UDF satisfies the Purity and the Non-null property, respectively. Otherwise, the DFA concludes that the UDF satisfies none of these properties. For the Parallel Safe property, the final analysis state indicates whether the UDF contains any operation that violates the Parallel Safe policy. If that is the case, the UDF satisfies the property. Otherwise, the DFA concludes that the UDF does not satisfy the property. For the Returns Null On Null property, the final analysis state contains a set of input variables that pass equality checks against the NULL value, which indirectly suggests that for each of these variables, once it evaluates to NULL, the UDF returns NULL. If all input parameters satisfy this criterion, we conclude the UDF satisfies the property. Otherwise, the DFA concludes that the UDF does not satisfy the property.

5 EVALUATION

To evaluate the usefulness of LAMBDA, we investigated the following questions:

RQ1. How effective and efficient is LAMBDA? (§5.3)
RQ2. How do UDF properties improve the DBMS performance? (§5.4)
RQ3. How does LAMBDA compare against other techniques for improving UDF performance? (§5.5)

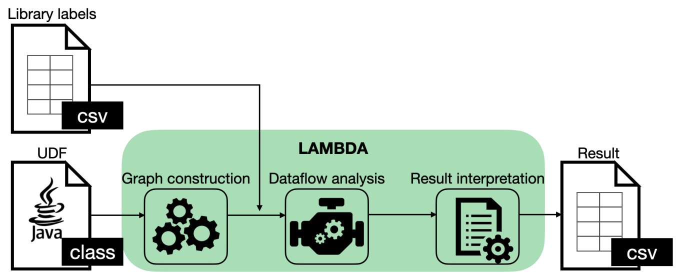


Figure 5: High-level overview of LAMBDA.

5.1 Implementation

We implement LAMBDA on top of Sparta [1], an extensible abstract interpretation framework for developing efficient DFA. The framework provides abstractions for us to implement dataflow facts, flow functions, join operators for each UDF property, and engines for performing DFA (*i.e.*, fixpoint iterator).

Architecture. Figure 5 shows the architecture of LAMBDA. It takes as input the UDFs that are in the form of Java class files. LAMBDA first constructs control flow graphs of the UDFs and then performs dataflow analysis on top of them. At the completion of the analysis, LAMBDA analyzes the final analysis state and translates it into a boolean value that indicates whether the UDF satisfies the property. The output of LAMBDA is stored in a CSV file that indicates whether input UDFs satisfy the property.

Defining shortcuts. LAMBDA performs interprocedural analysis by analyzing callees and storing the result as function summaries. However, for external libraries, given their large size and complex dependencies, we decided to use manual annotations. LAMBDA provides an interface for this purpose. Users can provide a CSV file that defines certain *shortcut rules* for a given UDF property. The CSV file contains 2 columns: the first and the second column specifies the name of the library call and whether it satisfies the property, respectively. During the dataflow analysis, if LAMBDA encounters a library call that does not have a *shortcut rule*, LAMBDA assumes it generates a $\top$ dataflow fact. Taking Purity analysis as an example, a $\top$ dataflow fact implies that the return value of the library call *may* be impure. We annotated 192 external library calls because they are used by UDFs that we studied. We annotated them by understanding their documentation and usage examples.

Extensibility. LAMBDA can be extended to infer additional UDF properties. Supporting a new UDF property only requires 3 items: ❶ a lattice that represents the abstraction of the program information; Notably, other UDF properties may require a lattice that has a different shape than the ones presented in this paper. For instance, the lattice that estimates the cardinality of the UDF output will share the same shape as the lattice that is used in range analysis. ❷ flow functions and join operators that define how to propagate the program information; ❸ shortcuts that model external libraries. Because LAMBDA currently hardcodes the first and the second items as Java classes, DBMSs developers may find it challenging to create them in order to support a new UDF property. In future work, we will investigate ways to measure the development difficulties of extending LAMBDA and design a more user-friendly interface for DBMS developers to support additional UDF properties.

5.2 Evaluation Setup

Our evaluation focused on a UDF benchmark that represents a realistic UDFs workload. The benchmark consists of 3 main components: the database, UDFs, and queries that invoke these UDFs. We have made the public version of the UDF benchmark available [9]. We ran all experiments on a server with two Intel(R) Xeon(R) E5649 CPUs (24 processors) and 236 GB RAM. Next, we present more details on how we construct the UDF benchmark.

Database. The benchmark uses a TPC-DS database [22] that is derived from the SQL-ProcBench [16]. The database contains 32 tables that take 3.8 GB in total. It is designed to model production workloads and is widely used for DBMS performance evaluation [23].

UDFs. We focus on scalar UDFs for our evaluations, which come from three sources. First, we reused the 20 UDFs from the case study (§2.3). These UDFs are not publicly available and are used for RQ1 evaluation. Second, in order to create publicly available UDFs that represent production UDFs, we manually created 5 UDFs that imitate the functionalities and programming patterns of UDFs from the first source. These 5 UDFs are for RQ2 and RQ3 evaluations [1]. Third, we randomly selected 10 out of 25 UDFs from SQL-ProcBench [16], an open benchmark for procedural workloads in DBMSs. Because UDFs from SQL-ProcBench are written in T-SQL, we manually translated these UDFs in Java. Compared with Java UDFs, UDFs derived from SQL-ProcBench UDFs have simpler functionalities and programming patterns: they are used to process tuples from database instances, by issuing queries to DBMSs and processing query results. On the contrary, Java UDFs in our benchmark have more varieties in terms of functionalities and complexities. In total, our UDF benchmark has 35 Java UDFs, among which 30 are used for RQ1, 15 are used for RQ2 and RQ3, and 20 are used for our case study on the proprietary platform.

Queries. For RQ2 and RQ3 evaluation, we constructed realistic UDF-invoking queries by adding *appropriate* UDF calls to a set of *base queries* that are realistic. We achieved this in three steps. ❶ We collected 15 *base queries* by randomly sampling queries from the TPC-DS benchmark. Queries from the benchmark are representative of production queries [23]. ❷ In order to know how to properly add UDF calls to base queries, we distilled common UDF invocation patterns from SQL-ProcBench:

```
Q:  SELECT projection FROM Table-1 ..... Table-k
       WHERE predicate
projection = Column-1 ..... UDF-call
predicate = Term-1 AND Term-2 AND ..... UDF-term
UDF-term = UDF-call Compare Constant | UDF-call In Constantset
Constant = Varchar | Integer | Decimal | Date | Null
Constantset = {Varchar} | {Integer} | {Decimal} | {Date}
```

There, we find UDFs are called in two places: predicate and projection. For invocations within the predicate, the UDF results are compared against either a constant variable or a set of constant variables for filtering purposes. For invocations within the projection, the UDF results are used as a column in the result table. ❸ We added *appropriate* UDF calls to *base queries* by referencing the UDF invocation patterns just learned. We assumed a UDF is *appropriate* to call only if the columns that the UDF should be called upon are available in the *base query*, by taking the UDF semantics into consideration. To limit the query complexity, we used only 1

[1]We did not use them for RQ1 in order to avoid evaluation bias

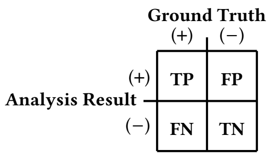


Figure 6: Taxonomy of analysis results (+ means property violations).

UDF for each UDF-invoking query. In order to guarantee the UDF-related predicates are realistic, we compared the UDF result against meaningful values that are randomly sampled either from the database or the UDF's source code. In this manner, we constructed 62 UDF-invoking queries that cover 15 UDFs.

5.3 RQ1 — Efficacy and Efficiency

We studied whether LAMBDA can correctly analyze the four UDF properties defined in §3. In other words, we examined whether LAMBDA can detect all violations of UDF property specifications. We focused on 30 UDFs from the benchmark. The evaluation has 3 steps. First, we gathered the ground truths. To do this, we performed a manual analysis on each UDF and checked whether it violates any property specification. Second, we evaluated LAMBDA on these UDFs and compared the analysis results with the ground truth. We classified the result according to the taxonomy presented in Figure 6 into four categories: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). We reported the number of TP, TN, and FP in Table 3. Notably, we did not report the number of FN in the table, because LAMBDA is designed to generate 0 FN (by conservatively reporting possible property violations).

The first observation is that LAMBDA accurately analyzed 3 UDF properties: Non-null, Returns Null On Null, and Parallel Safe. LAMBDA reported 0 false positives and negatives for these properties. One possible explanation is that the analysis of these properties does not demand comprehensive and complicated modeling of UDF behaviors. For instance, the Returns Null On Null property only requires analyzing code that handles NULL inputs.

The second observation is that, while LAMBDA correctly analyzed most UDFs for the Purity property, it generated four false positives that missed UDFs that should be pure. This is because LAMBDA conservatively reports cases wherein usages of shared variables could possibly violate the property specification. Shared variables usage is a common programming practice we observed in production UDFs. As mentioned in §2.1, UDFs have access to class variables and may modify them whenever they are executed. As a result, if a UDF does not properly *sanitize* these member variables before using them, it can violate the specification of Purity. Based on this observation, LAMBDA always checks whether a given UDF *sanitizes* shared member variables, by checking for the presence of certain library calls. However, as the evaluation results demonstrated, this check causes false positives for some special cases: (1) when shared member variables are constants (*e.g.*, never rewritten by the UDF) (2) when shared member variables only need *sanitization* under certain path conditions. It is feasible to make LAMBDA support both cases, by implementing a constant analysis with respect to the member variables and a symbolic analysis of the path constraints, respectively.

Table 3: Efficacy of LAMBDA.

Property	TP	TN	FP
Purity	11	15	4
Non-null	22	8	0
Returns Null On Null	13	17	0
Parallel Safe	17	13	0

We also examined whether LAMBDA can catch developers' annotation mistakes for production UDFs. In particular, as demonstrated in §2.3, 5 UDFs are incorrectly labeled by users as pure. LAMBDA was able to determine them all as impure. We summarize 2 behaviors that make these UDFs impure: (1) UDFs leverage exception and try blocks for input preprocessing, wherein system information is introduced to replace invalid values; (2) UDFs read from variables that are mutable across UDF invocations. We reported all annotation bugs to developers. In addition to acknowledging all reported cases, they also provided insights about improving LAMBDA. The response we received is along the lines of: "These are really helpful and interesting findings. Given that users may not know well about the specification of the UDF property, another valuable information that the tool can provide is about why the UDF violates this property." As part of our future work, we plan to investigate ways to add this suggested feature and incorporate LAMBDA into their proprietary data-processing platform.

We also tracked how much time LAMBDA took to run, by dividing the total analysis time by the number of UDFs being analyzed. On average, it took 34.9 milliseconds for LAMBDA to analyze 1 UDF. Because the LAMBDA runtime is negligible compared to that of processing UDF-invoking queries, DBMSs users can integrate LAMBDA with the existing data processing pipeline without significant performance regressions. When a UDF has been modified, LAMBDA users can comfortably run it again to make sure the change does not break existing UDF property annotations.

LAMBDA was able to infer four UDF properties efficiently and without reporting false negatives. The properties considered posed different challenges for the analysis. Given these results, LAMBDA seems suitable for integration into existing DBMSs.

5.4 RQ2 — Performance Improvement

We studied the impact of leveraging UDF properties on query performance. We focused our evaluation on PostgreSQL (v14.3). We focused on 4 UDF properties that are defined in §3. We break our evaluation into 2 parts. First, for each UDF-invoking query from our benchmark (referred to as the *regular* query), we tried to create its *optimized* versions by leveraging the UDF properties that the UDF satisfies (§5.4.1). Second, for each *regular* and *optimized* query pair (created by leveraging a given UDF property), we ran both queries on the benchmark database and compared their query execution time. If the *optimized* query runs faster than the *regular* query, it demonstrates that leveraging the UDF property improves query performance. We present the evaluation result in §5.4.2.

5.4.1 Query Pair Construction. For queries that invoke pure UDFs, we created their *optimized* versions that simulate the effect of reusing the UDF results. In particular, the *optimized* query calls an *memoized* UDF that stores the UDF results and returns the cached result when the same inputs occur again. For queries that invoke Non-null UDFs and have comparisons between NULL and the UDF

Table 4: Query runtime performance statistics.

Property	t_reg(s)		t_opt (s)		# of queries	
	avg	std	avg	std	speedup	¬speedup
Purity	25.7	38.6	11.4	14.2	23	39
Non-null	29.0	47.1	27.9	46.9	4	7
Returns Null On Null	25.3	37.6	25.0	38.2	11	38
Parallel Safe	31.4	41.3	30.2	42.2	17	33 (7)

result, we created *optimized* queries by removing the comparisons. For queries that invoke Returns Null On Null UDFs, we created *optimized* queries that invoke the same UDFs but have annotations that allow PostgreSQL to leverage the Returns Null On Null property. For queries that invoke Parallel Safe UDFs, we created *optimized* queries that invoke the same UDFs but have annotations that allow PostgreSQL to process UDFs with parallelism. In this manner, we created 62, 11, 49, and 50 *regular* and *optimized* query pairs for Purity, Non-null, Returns Null On Null, and Parallel Safe, respectively.

5.4.2 Query performance result. For each *regular* and *optimized* query pair, we ran each query 5 times and computed the average of their query response time T_{reg} and T_{opt} respectively[2]. For each UDF property and its associated query pairs, we reported the average and standard deviation of their query response time in Table 4. For each query pair, we also computed the percentage of the query execution time reduction after leveraging the UDF property using the formula $\left(\left(T_{reg} - T_{opt}\right) / T_{reg}\right) \times 100\%$. The possible value for the percentage of reduction falls into the range $(-\infty, 100\%)$. Figure 7 presents the percentage of time reduction results for all UDF properties. In order to account for the effect of environmental noises on time measurement, we only reported performance regression and improvement if the percentage of reduction is lower than -10% and higher than 10%, perspectively. We reported the number of queries with and without performance improvements in Table 4, whereas the number in parentheses indicates the number of regressions.

The first observation is that leveraging UDF properties improved query performance in some cases. In particular, Purity improved query performance in 23 out of 62 cases (time reduction ranging from 10% to 99%), Non-null improved query performance in 4 out of 11 cases (time reduction ranging from 12% to 40%), Returns Null On Null improved query performance in 11 out of 49 cases (time reduction ranging from 10% to 58%), and Parallel Safe improved query performance in 17 out of 50 cases (time reduction ranging from 10% to 74%).

The second observation is that leveraging UDF properties does not always improve query performance. In particular, Purity, Non-null, Returns Null On Null, Parallel Safe did not improve query performance in 39 out of 62, 7 out of 11, 38 out of 49, and 26 out of 50 cases, respectively. After examining the query execution plans, we identified three possible reasons for this. First, the significance of speedup depends on the number of UDF calls saved. For Returns Null On Null, because it only eliminates UDF invocations that have NULL inputs, the magnitude of speedup depends on the number of NULL values within tuples that UDFs are called upon. In our benchmark database, because each table column contains only a small amount (*i.e.*, <10%) of NULL value, the number of UDF invocations that can be optimized away by Returns Null On Null are limited. Second, the significance of

[2]We made sure the query cache is not used

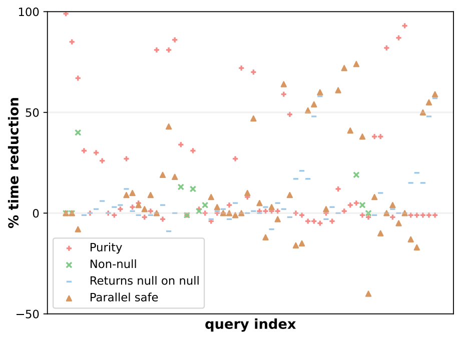


Figure 7: Percentage of reduction of query response time by leveraging UDF properties (higher is better).

speedup also depends on the query complexity. For complex queries that contain computationally expensive relational operators (*e.g.*, , joins, aggregation, and subqueries), leveraging Purity and Non-null do not lead to speed-up, even when they save UDF calls. This observation aligns well with findings from the previous paper [16]. However, that paper also points out that complex queries inevitably begin to suffer from the performance bottleneck of UDFs when the size of datasets increases to a certain extent, which highlights the performance benefits of leveraging Purity and Non-null on large datasets. Third, it is ultimately the DBMS's decision whether to choose an alternative execution plan that leverages the UDF properties. For Parallel Safe and some of its associated queries, PostgreSQL decides to keep the original execution plans, which results in no performance difference among query pairs.

Thirdly, we found that leveraging Parallel Safe introduces performance regressions in 7 out of 50 cases (time increase ranging from 10% to 1001%). These regressions are caused by the query optimizer's incorrect decision to execute the query with parallelism, which turns out to be slower than the default (sequential) execution mode. These regressions highlight the limitation of the query optimizer in computing the optimal plan to process UDFs. One root cause is that the query optimizer does not know the execution cost of UDFs [25]. To address this limitation, in future work, we will investigate ways to estimate UDF execution cost.

5.4.3 Analysis Overhead vs Query Performance Improvement. Based on the statistics from Table 4, we calculated that the average reduction of query response time after leveraging UDF properties is 5.7 seconds. Given that the average analysis time for LAMBDA is 34.9 milliseconds, we expect that the benefits of running LAMBDA are more likely to outweigh its analysis overhead. In addition, because LAMBDA only needs to run after updates of UDFs or property specifications, it is not necessary to run LAMBDA before each query invocation. We expect LAMBDA to be used as an offline analysis tool (which is also confirmed by developers). As a result, the analysis cost of LAMBDA can be amortized over a period of time that depends on the frequency of UDF updates.

UDF properties differ in how and to what extent they improve query performance. Leveraging Parallel Safe can trigger performance regressions, which highlights opportunities for improving future versions of DBMSs. The query performance improvement after leveraging UDF properties is greater than the analysis overhead of LAMBDA, which supports the applicability of LAMBDA.

5.5 RQ3 — Comparative Analysis

Recent research efforts, such as FROID [25], improve UDF performance by transforming UDFs that are implemented in PL/SQL languages into scalar subqueries. To answer RQ3, we evaluated FROID on the UDF benchmark and qualitatively compared its UDF optimization results against LAMBDA. We used the same UDFs and queries as RQ2. Notably, we did not directly compare the runtime performance between FROID- and LAMBDA-optimized queries, because FROID is only available on SQLSERVER (v15.0.2) and not on PostgreSQL. Our evaluation focused on 2 aspects: (1) how many UDFs FROID can inline, and (2) how many UDF-calling queries FROID can optimize.

INLINABLE UDFS. Among 15 UDFs from our benchmark, 7 UDFs can be inlined by FROID and they are all derived from the SQL-ProcBench. 8 UDFs cannot be inlined by FROID, because they violate conditions that UDFs must satisfy in order to guarantee that rewriting them in SQL preserves the original semantics [8]. An example of such a condition is that the UDF can not have more than 1 return statement. This experiment demonstrates that FROID cannot apply to all UDFs: because UDFs are designed to extend the expressiveness of the SQL query, it is very likely that they contain logic and imperative constructs that can not be translated back into SQL. On the other hand, we found LAMBDA helps optimize more UDFs than FROID: LAMBDA determined that each of the 15 UDF satisfies at least 1 property that can improve query performance. One possible explanation behind the different coverage of UDFs is that LAMBDA is more language-agnostic than FROID: it does not prevent UDFs from using any programming constructs.

OPTIMIZATION OF UDF-INVOKING QUERIES. Next, we inspected whether FROID can optimize queries that invoke UDFs that are inlinable. For our benchmark, there are 26 queries that invoke the 7 UDFs that FROID can inline. In order to determine whether FROID helps SQLSERVER optimize these queries, we ran each of these queries on our benchmark database two times, with and without enabling FROID, and compared the query execution plans and execution times. Surprisingly, FROID only optimized 1 out of 26 queries. For the other 25 queries, enabling FROID did not lead to a different execution plan. On the other hand, LAMBDA helped PostgreSQL compute better execution plans for 13 out of 26 queries. There are 2 possible explanations behind the fact that FROID cannot optimize queries that invoke UDFs, even though the UDFs can be transformed. First, in order to make sure the translation process does not break the original semantics of the query, FROID limits clauses wherein UDFs are invoked [8]. For instance, FROID cannot take effect when a query invokes it in ORDER BY or GROUP BY clauses. Second, and somehow inevitably, transforming UDFs into SQL subquery makes the query more complicated, which makes it more challenging for the DBMS to compute a more efficient query execution plan. To mitigate the problem, FROID restricts the size of the query to perform the UDF transformation. In turn, FROID does not take effect when the size of the query exceeds a certain threshold, which limits the space of UDF-invoking queries that FROID optimizes. We believe these findings demonstrate that LAMBDA is a valuable complementary technique to FROID.

LAMBDA optimized more UDF-invoking queries than FROID. Two reasons for this better performance are that LAMBDA does not restrict (1) the usage of imperative language constructs and (2) the structure and size of UDF-invoking queries.

6 RELATED WORK

UDF TRANSLATION. Multiple existing research efforts focus on transforming complete UDFs into equivalent SQL subqueries [15, 17–19, 25, 28]. While these techniques improve UDF performance by eliminating UDFs, they are limited to a subset of imperative language constructs and can lead to performance regression for complex UDFs and queries. In contrast, LAMBDA is not limited by imperative language constructs and can be extended to support additional imperative languages. Because our experiment demonstrates that parallel execution of UDFs can lead to performance regressions, we plan to investigate additional UDF properties that can help DBMSs with parallelism.

UDF COMPILATION. Another line of research focuses on compiling queries that invoke UDFs into a unifying IR to enable optimizations across language boundaries [11, 14, 26]. However, because these approaches treat UDFs as black boxes, they are limited to a subset of optimizations. LAMBDA can enhance these systems by providing UDF properties that enable more optimization opportunities.

7 CONCLUSION

We presented LAMBDA, a new approach for improving DBMS performance in the presence of UDFs. The key idea behind LAMBDA is to infer properties of UDFs and provide these properties as annotations to the DBMSs. Specifically, we focused on four commonly used UDF properties: PURITY, NON-NULL, RETURNS NULL ON NULL, and PARALLEL SAFE. To assess our approach, we implemented LAMBDA and evaluated it on a benchmark derived from production query workloads and real-world UDFs, with promising results. LAMBDA was able to infer the targeted four UDF properties from 30 production UDFs, and the extracted properties allowed the DBMS to improve query performance considerably, with time reductions ranging from 10% to 99%. Furthermore, LAMBDA generated 0 false negatives, and the average analysis time was 34.9 milliseconds for each UDF. We also compared LAMBDA to FROID, a state-of-the-art framework for improving UDF performance based on transforming UDFs into semantically equivalent SQL. LAMBDA was able to optimize UDFs and queries in cases in which FROID was unable to handle the UDFs. We believe that these results provide initial, yet strong evidence that LAMBDA can be a useful technique for improving UDF performance. In future work, we plan to incorporate LAMBDA into an existing DBMS and improve our technique based on our findings. Our current results, for instance, highlight that leveraging PARALLEL SAFE property may lead to performance regressions. We will also investigate ways to infer other properties that help DBMSs process UDFs with parallelism.

ACKNOWLEDGMENTS

This work was partially supported by NSF (grants CCF-0725202, IIS-1908984, IIS-2238431), DOE (contract DE-FOA-0002460), Alibaba Innovative Research (AIR) Program, and gifts from Adobe, Cisco, Facebook, Google, IBM Research, Intel, and Microsoft Research.

REFERENCES

[1] 2019. Sparta. https://engineering.fb.com/2019/02/20/open-source/sparta/.

[2] 2022. Common subexpression elimination explained. https://learn.microsoft.com/en-us/sql/analytics-platform-system/common-sub-expression-elimination?view=aps-pdw-2016-au7.

[3] 2022. CREATE MATERIALIZED VIEW AS SELECT (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/statements/create-materialized-view-as-select-transact-sql?view=azure-sqldw-latest.

[4] 2023. Froid vs Freud: Scalar T-SQL UDF Inlining. https://www.nikoport.com/2019/07/16/froid-vs-freud-scalar-t-sql-udf-inlining/.

[5] 2023. IBM Db2 UDF. https://www.ibm.com/docs/en/db2-for-zos/11?topic=statements-create-function-inlined-sql-scalar.

[6] 2023. MySQL UDF. https://dev.mysql.com/doc/refman/8.0/en/function-optimization.html.

[7] 2023. PostgreSQL UDF. https://www.postgresql.org/docs/current/sql-createfunction.html.

[8] 2023. Scalar UDF Inlining. https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining?view=azuresqldb-current&viewFallbackFrom=sql-server-2017.

[9] 2023. Supplementary material. https://github.com/sub-helper/lambda_artifact

[10] Anonymity. 2023. The platform name is anonymous for the review.

[11] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. In *VLDB*. 1466–1477.

[12] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away. arXiv:1909.03291 [cs.DB]

[13] Sofoklis Floratos, Ahmad Ghazal, Jason Sun, Jianjun Chen, and Xiaodong Zhang. 2021. DBSpinner: Making a Case for Iterative Processing in Databases. In *ICDE*. 2399–2410.

[14] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient execution of polyglot queries. In *VLDB*. 196–210.

[15] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 559–573.

[16] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. In *VLDB*. 1378–1391.

[17] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting pandas in a box. In *CIDR*.

[18] Denis Hirn and Torsten Grust. 2021. One with recursive is worth many GOTOs. In *SIGMOD*. 723–735.

[19] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends.. In *CIDR*.

[20] John B Kam and Jeffrey D Ullman. 1976. Global data flow analysis and iterative algorithms. In *JACM*. 158–171.

[21] John B Kam and Jeffrey D Ullman. 1977. Monotone data flow analysis frameworks. In *Acta informatica*. 305–317.

[22] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *VLDB*. 1049–1058.

[23] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis.. In *VLDB*. 1138–1149.

[24] Miodrag Potkonjak, Mani B Srivastava, and Anantha P Chandrakasan. 1996. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. In *IEEE TCAD*. 151–165.

[25] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. In *VLDB*. 432–444.

[26] Maximilian E Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql. In *SSDBM*. 1–12.

[27] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. In *TOPLAS*. 181–210.

[28] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. In *OOPSLA*. 1–26.