BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid

Karthik Ramachandra
Microsoft Research India
karam@microsoft.com

Kwanghyun Park
Microsoft Gray Systems lab
kwpark@microsoft.com

ABSTRACT

Relational DBMSs allow users to extend the standard declarative SQL language surface using User Defined Functions (UDFs) that implement custom behavior. While UDFs offer many advantages, it is well-known amongst practitioners that they can cause severe degradation in query performance. This degradation is due to the fact that state-of-the-art query optimizers treat UDFs as black boxes and do not reason about them during optimization.

We demonstrate Froid, a framework for optimizing UDFs by opening up this black box and exposing its underlying operations to the query optimizer. It achieves this by systematically translating the entire body of an imperative multi-statement UDF into a single relational algebraic expression. Thereby, any query invoking this UDF is transformed into a query with a nested sub-query that is semantically equivalent to the UDF. We then leverage existing sub-query optimization techniques and thereby get efficient, set-oriented, parallel query plans as opposed to inefficient, iterative, serial execution of UDFs.

We demonstrate the benefits of Froid including performance gains of up to multiple orders of magnitude on real workloads. Froid is available as a feature of Microsoft SQL Server 2019 called 'Scalar UDF Inlining'.

PVLDB Reference Format:
Karthik Ramachandra and Kwanghyun Park. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *PVLDB*, 12(12): 1810-1813, 2019.
DOI: https://doi.org/10.14778/3352063.3352072

1. INTRODUCTION

User Defined Functions (UDFs) provide an elegant, powerful abstraction that enables application developers embed custom business logic inside a database. In fact, UDFs and views are the primary ways to reuse code in database applications. UDFs support familiar imperative programming constructs, and hence are often preferred by non SQL experts. Complex business rules are often easier to express as well as understand when written using imperative UDFs rather than complex SQL.

Although relational database systems support UDFs, they are almost always accompanied by a word of caution: The advantages of UDFs today come with a huge performance penalty. As a result, users are advised to avoid UDFs when performance is critical to their use cases, thereby compromising on modularity, reusability and readability of application code.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352072

```
--Query:
SELECT C_NAME, dbo.service_level(c_custkey)
FROM CUSTOMER

-- UDF definition:
create function service_level(@ckey int)
returns char(10) as
begin
    declare @total float;
    declare @level char(10);

    select @total = sum(o_totalprice)
    from orders where o_custkey = @ckey;

    if(@total > 1000000)
        set @level = 'Platinum';
    else if(@total > 500000)
        set @level = 'Gold';
    else
        set @level = 'Regular';

    return @level;
end
```

Figure 1: Query invoking a UDF, and the UDF definition. Example taken from Simhadri et. al.[7].

1.1 Illustrative Example

Consider the scalar UDF (UDFs that return a scalar value are called scalar UDFs) shown in Figure 1. Given a customer key, the UDF *service_level* determines the service category for that customer. It arrives at the category by first computing the total price of all orders placed by the customer using a SQL query, and then uses an IF-ELSE logic to decide the category based on the total price. A simple query that invokes this UDF is given in Figure 1 which lists all customers and their service level. This UDF is very handy because it can now be used in multiple queries, and if the threshold values need to be updated, or a new category needs to be added, the change must be made only in the UDF.

```
select c.c_custkey,
    case when e.total > 1000000 then 'Platinum'
       when e.total > 500000 then 'Gold'
       else 'Regular'
    end
from customer c left outer join
  (select o_custkey, sum(o_totalprice) as total
   from orders
   group by o_custkey) e
on c.c_custkey=e.o_custkey;
```

Figure 2: A manually written query (without UDFs) that is equivalent to the query in Figure 1. Example taken from Simhadri et. al.[7].

Figure 2 shows an alternate implementation of the same requirement of computing the service category for all customers. This implementation is entirely in declarative SQL form, and does not invoke any UDFs. From Figure 2, it is obvious to the reader that (a) it is harder to understand the behavior, especially for non-expert SQL users, and (b) the computation of service category is not reusable if the same behavior is required in another query.

However, if the queries in Figure 1 and Figure 2 are executed even on a moderately large dataset, the difference in both end-to-end execution time as well as resource utilization would stand out. The query with UDF (Figure 1) would run observably (orders of magnitude) slower than the query in Figure 2 on a RDBMS. This forces users to either give up performance if they need the benefits of UDFs, or avoid using UDFs if performance is important.

1.2 Challenges in UDF optimization

Some of the key reasons for poor performance of UDFs are as follows:

Iterative execution: UDFs are invoked in an iterative manner, once per qualifying tuple. This incurs additional costs of repeated context switching due to function invocation. Especially, UDFs that execute SQL queries in their body (common in real workloads) are severely affected.

One could argue that known compiler techniques could be used to optimize imperative UDFs by compiling them into efficient native code. Although native compilation makes UDFs faster, the benefits are limited as the query still invokes the UDF for each tuple. We show how Froid removes this fundamental limitation and hence combining Froid with native compilation leads to more gains [5].

Lack of costing: Query optimizers treat UDFs as inexpensive black-box operations. This is a crucial cause of bad plan choices in cases where scalar operations are arbitrarily expensive, which is often true for scalar UDFs.

Limitation on parallelism: Currently, SQL Server does not use intra-query parallelism in queries that invoke UDFs. Note that it is not trivial to parallelize arbitrary UDFs, especially those that contain SQL queries in their body. This is because each query inside the UDF might also use parallelism, and it is not straightforward to decide the best strategy to share threads across these.

1.3 Solution overview

Froid [5] is an extensible framework for optimizing imperative UDFs in a relational DBMS. The purpose of Froid is to enable developers to use the abstraction of UDFs without compromising on performance. In the above example, using

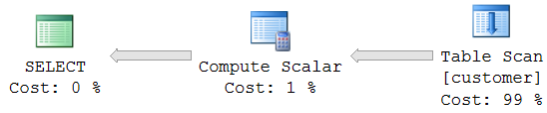

Figure 3: Query execution plan for query in Figure 1 (without Froid)

Froid on Figure 1 would yield the same performance as the SQL query in Figure 2. Thereby, we get all the benefits of UDFs, along with the performance of declarative SQL.

Froid introduces a radically different approach to evaluating imperative functions that results in performance improvements of up to multiple orders or magnitude over the existing state of the art. Froid was recently announced as a feature of Microsoft SQL Server 2019 [8] called **"Scalar UDF Inlining"** [2]. The techniques underlying Froid can be integrated into any RDBMS which has sub-query optimization techniques built into it.

At the core of Froid is a technique that can automatically transform an entire multi-statement UDF into an equivalent relational algebraic expression. This form is now amenable to cost-based optimization and results in highly efficient, set-oriented, parallelizable execution strategies as opposed to inefficient, iterative, serial execution of UDFs. The work of Simhadri et. al [7] describes decorrelation techniques for UDF invocations. Froid improves upon those ideas to build a complete industrial-strength optimization framework.

Froid essentially transforms entire UDFs into SQL and therefore, queries invoking UDFs are transformed into queries with nested sub-queries. This enables the query optimizer to use well-known sub-query decorrelation techniques including Magic decorrelation [6] and others [3, 4]. *Since Froid opens up the **"Black box"** UDFs to the query optimizer, and enables the use of **"Magic"** and other optimization techniques, we name this demonstration as **"BlackMagic"**.*

2. FEATURES OF FROID

Froid offers a comprehensive solution to the performance problems of UDFs – it overcomes all the current drawbacks in UDF evaluation described in Section 1.2. The benefits due to Froid can be demonstrated using two query plans for the query in Figure 1. The query plan with Froid disabled is given in Figure 3, and the plan with Froid enabled is given in Figure 4. We first define the scope of Froid's applicability and then describe its features.

Supported UDFs: Froid currently supports the following imperative constructs in scalar UDFs.

- **DECLARE, SET:** Variable declaration and assignments.
- **SELECT:** SQL query with multiple variable assignments.
- **IF/ELSE:** Branching with arbitrary levels of nesting.
- **RETURN:** Single or multiple return statements.
- **UDF:** Nested/recursive function calls.
- **Others:** Relational operations such as EXISTS, ISNULL.

Froid does not impose restrictions on the size/depths of UDFs and complexity of queries that invoke them. However, there are certain cases where we block inlining; they are discussed in [5]. Froid currently supports T-SQL UDFs, but the underlying techniques are language-agnostic, and extensible to other languages.

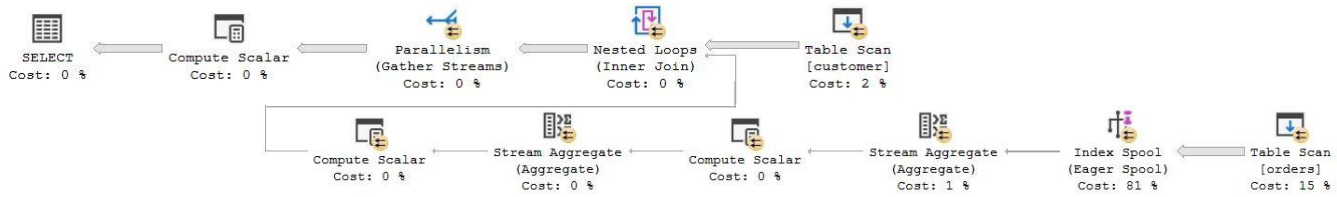

Figure 4: Query execution plan for query in Figure 1 (with Froid)

2.1 Set-oriented evaluation of UDFs

The *Compute Scalar* operator in Figure 3 represents UDF execution. From Figure 3, it is clear that the UDF is invoked and executed iteratively, once per qualifying tuple emitted by the FROM clause. Iterative plans are more inefficient in cases where the UDF in turn executes SQL queries. In contrast, with Froid, the operations in the UDF are evaluated in a set-oriented manner as shown in Figure 4. This turns out to be one of the main reasons for performance improvements observed in our experiments.

Note that vectorization, JIT compilation, or native compilation cannot yield these benefits for UDFs. This is because these techniques optimize the UDF definition independent of the query invoking it and therefore, correlated execution of UDFs is not eliminated. Froid, on the other hand, performs a combined optimization of the UDF along with the query invoking it, and therefore is able to decorrelate UDF invocations and achieve set-oriented execution.

2.2 Inferring implicit relational operations

Imperative programs typically go through a compiler that performs several optimizations. It can be argued that we could use well-known compiler optimization techniques to optimize UDFs as well. However, UDFs executing in a relational database present a different set of challenges and opportunities. Relational operations such as JOINs and GROUP BYs could be implicit in a UDF, and a traditional compiler will not be able to infer such operations.

Database users who are not SQL experts often express JOINs and other relational operations implicitly using UDFs. For instance, in Figure 1 the customer key (*c_custkey*) passed in as a parameter to the UDF is used to look up all orders of that customer and compute the sum of their price (*o_totalprice*). The observant reader can immediately realize that this is in fact a JOIN between CUSTOMER and ORDERS which is hidden through function invocation. Without inlining, the optimizer has no choice other than an iterative plan with expensive function invocation overheads.

Froid can infer implicit relational operations and make them explicit, as shown in Figure 4 – which performs a JOIN between CUSTOMER and ORDERS. This empowers the optimizer to choose any implementation of these relational operations based on cardinality and cost estimates. The optimizer has also inferred a GROUP BY on *o_custkey* (indicated by the *Stream Aggregate* operators in Figure 4) which was implicit in the UDF. In most cases, the shape of the plan due to Froid (e.g. Figure 4) would be similar to that of a manually written equivalent query without UDFs.

2.3 Costing of operations inside UDFs

Query optimizers treat UDFs as inexpensive black-box operations. During optimization, only relational operators are costed, while scalar operators are not. Prior to the introduction of scalar UDFs, other scalar operators were generally cheap and did not require costing. A small CPU cost added for a scalar operation was enough. This inadvertent simplification is a crucial cause of bad plan choices in cases where scalar operations are arbitrarily expensive, which is often true for scalar UDFs. Thanks to Froid, expensive operations inside the UDF are now visible to the optimizer (as Figure 4 shows), and are hence costed. This greatly improves plan quality, and hence performance.

2.4 Exploiting intra-query parallelism

As mentioned earlier, parallelizing arbitrary UDFs is non-trivial. For instance, consider the UDF in Figure 1 which internally invokes an SQL query. Each such query may itself use parallelism, and therefore, the optimizer has no way of knowing how to share threads across them, unless it looks into the UDF and decides the degree of parallelism for each query within (which could potentially change from one invocation to another). With nested and recursive UDFs, this issue becomes even more complex.

With Froid, this limitation no longer holds since all operations inside the UDF are now exposed to the optimizer, which can then come up with highly parallel plans. In Figure 4, we observe the use of parallelism for almost all operators (the double-arrow symbol superposed on an operator indicates the use of parallelism for that operator).

2.5 Compiler optimizations

Froid's approach not only overcomes current drawbacks in UDF evaluation, but also adds a bonus: with no additional implementation effort, it brings to UDFs the benefits of several optimizations done by an imperative language compiler. The currently supported optimizations are (a) Dynamic Slicing, (b) Constant Folding, (c) Constant Propagation, (d) Dead code elimination, and (e) Some forms of common sub-expression elimination. Depending upon the UDF definition, these optimizations kick in automatically. Their effects can be observed/demonstrated in the resulting query plan. Details and an example can be found in [5].

3. OVERVIEW OF TECHNIQUES

We now give a brief overview of the novel techniques underlying Froid. As described earlier, a commonly used execution strategy for UDFs is to evaluate them iteratively, similar to the correlated evaluation strategy for nested sub-queries [3]. In query optimization literature, this is considered to be an inferior strategy in general, as it involves per-row processing instead of set-oriented processing.

Optimization of sub-queries is well-studied [4, 3, 1], and database systems employ powerful techniques for evaluation of queries with nested sub-queries [3]. These techniques are

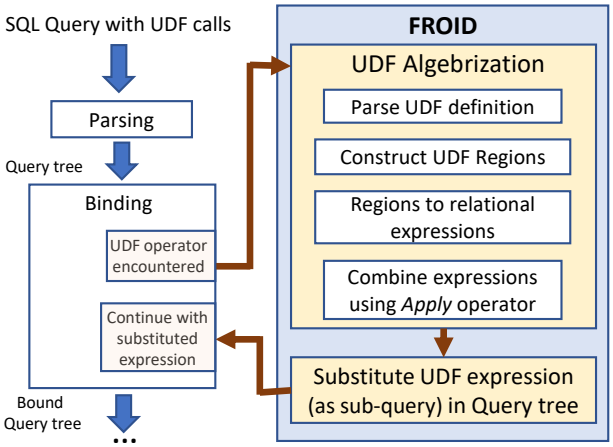

Figure 5: Overview of the Froid framework

able to transform correlated sub-queries into joins, thereby enabling set-oriented plans instead of iterative evaluation of sub-queries. This has been seen to result in efficient query plans, leading to significant improvement in performance.

Simhadri et. al. [7] show that the techniques designed to optimize nested sub-queries can be extended to optimize UDFs. They show how UDFs can be seen as complex sub-queries that are defined using a mix of imperative constructs and SQL, and can be brought into the well-studied framework of optimizing nested sub-queries. Froid borrows its intuition from [7] and demonstrates how such a technique can be integrated into an industrial strength database system. A discussion on related work can be found in [5].

The UDF Inlining Technique

Figure 5 depicts the high-level approach of Froid, consisting of two phases: UDF algebrization followed by substitution. As a part of SQL query binding (which includes validating referenced objects and loading metadata), if a UDF operator is encountered, the control is transferred to Froid, and UDF algebrization is initiated.

The goal of UDF algebrization is to build a single relational expression which is semantically equivalent to the UDF. This involves transforming imperative constructs into equivalent relational expressions and combining them in a way that strictly adheres to the procedural intent of the UDF. Froid achieves this goal by combining expressions for every program region using the *Apply* operator [3].

This resulting expression is then substituted, or *inlined* in the query tree of the calling query in place of the UDF operator. This query tree with the substituted UDF expression is bound using the regular binding process. If references to other (nested) UDF operators are encountered, the same process is repeated. This transformation finally results in a bound query tree, which forms the input to query optimization. Details can be found in [5].

We call this semantics-preserving transformation as *unnesting* or *inlining* of the UDF into the calling query. *Although we use the term inlining here, note that it is fundamentally different from inlining in imperative programming languages.*

Evaluation: We have conducted a detailed experimental evaluation of Froid to measure its applicability as well as impact. The details are available in [5]. We have observed that Froid results in significant performance gains across benchmarks and real customer workloads, and results in negligible compile-time overheads. Froid also results in significant resource savings, which are highly valuable in cloud scenarios.

4. DEMONSTRATION

Froid will be demonstrated in Microsoft SQL Server 2019 [8]. Our demonstrations will showcase various features of Froid and dive into the details of the underlying transformations. We will use the TPC-H benchmark and 3 real customer workloads (anonymized) to illustrate the benefits. The demo will be made highly interactive by allowing users to write their own UDFs and play with Froid's transformations and analyze the resulting query execution plans and performance.

The query plans with and without Froid will be visualized as shown in Figures 3 and 4 using the SQL Server Management Studio GUI tool. A detailed comparison of these plans will highlight the novelty and effectiveness of Froid. We will also show the intermediate steps that depict the transformation from Figure 3 to Figure 4. Users will be able to run these queries and observe the performance gains. We will also showcase the reduction in CPU time and disk IO.

5. CONCLUSION

We present Froid, a novel framework for optimizing UDFs in relational database systems. We describe several features of Froid and briefly outline the underlying techniques. To the best of our knowledge, Froid is the first industrial-strength framework that can optimize imperative multi-statement UDFs in a RDBMS by transforming them into relational expressions and inlining them into the calling query. Using real-world workloads, we demonstrate the features of Froid and the significant performance gains achieved. Readers are encouraged to download and use Froid, which is available as part of Microsoft SQL Server 2019 [8].

6. REFERENCES

[1] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution Strategies for SQL Subqueries. In *ACM SIGMOD*, 2007.

[2] Scalar UDF Inlining. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining.

[3] C. A. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[4] W. Kim. On Optimizing an SQL-like Nested Query. In *ACM Trans. on Database Systems, Vol 7, No.3*, 1982.

[5] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of imperative programs in a relational database. *PVLDB*, 11(4):432–444, 2017.

[6] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 450–458, Washington, DC, USA, 1996.

[7] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan. Decorrelation of user defined function invocations in queries. In *ICDE*, pages 532–543, March 2014.

[8] Microsoft SQL Server 2019. https://www.microsoft.-com/en-us/sql-server/sql-server-2019.