Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates

Surabhi Gupta
Microsoft Research India
t-sugu@microsoft.com

Sanket Purandare*
Harvard University
sanketpurandare@g.harvard.edu

Karthik Ramachandra
Microsoft Research India
karam@microsoft.com

ABSTRACT

Loops that iterate over SQL query results are quite common, both in application programs that run outside the DBMS, as well as User Defined Functions (UDFs) and stored procedures that run within the DBMS. It can be argued that set-oriented operations are more efficient and should be preferred over iteration; but from real world use cases, it is clear that loops over query results are inevitable in many situations, and are preferred by many users. Such loops, known as cursor loops, come with huge trade-offs and overheads w.r.t. performance, resource consumption and concurrency.

We present Aggify, a technique for optimizing loops over query results that overcomes these overheads. It achieves this by automatically generating custom aggregates that are equivalent in semantics to the loop. Thereby, Aggify completely eliminates the loop by rewriting the query to use this generated aggregate. This technique has several advantages such as: (i) pipelining of entire cursor loop operations instead of materialization, (ii) pushing down loop computation from the application layer into the DBMS, closer to the data, (iii) leveraging existing work on optimization of aggregate functions, resulting in efficient query plans. We describe the technique underlying Aggify, and present our experimental evaluation over benchmarks as well as real workloads that demonstrate the significant benefits of this technique.

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389736

*Work done during an internship at Microsoft Research India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389736

1 INTRODUCTION

Since their inception, relational database management systems have emphasized the use of set-oriented operations over iterative, row-by-row operations. SQL strongly encourages the use of set operations and can evaluate such operations efficiently, whereas row-by-row operations are generally known to be inefficient.

However, implementing complex algorithms and business logic in SQL requires decomposing the problem in terms of set-oriented operations. From an application developers' standpoint, this can be fairly hard in many situations. On the other hand, using simple row-by-row operations is often much more intuitive and easier for most developers. As a result, code that iterates over query results and performs operations for every row is extremely common in database applications, as we show in Section 10.2.

In fact, the ANSI SQL standard has had the specialized CURSOR construct specifically to enable iteration over query results[1] and almost all database vendors support CURSORs. As a testimonial to the demand for cursors, we note that cursors have been added to procedural extensions of BigData query processing systems such as SparkSQL, Hive and other SQL-on-Hadoop systems [30]. Cursors could either be in the form of SQL cursors that can be used in UDFs, stored procedures etc. as well as API such as JDBC that can be used in application programs [3, 31].

While cursors can be quite useful for developers, they come with huge performance trade-offs. Primarily, cursors process rows one-at-a-time, and as a result, affect performance severely. Depending upon the cardinality of query results on which they are defined, cursors might materialize results on disk, introducing additional IO and space requirements. Cursors not only suffer from speed problems, but can also acquire locks for a lot longer than necessary, thereby greatly decreasing concurrency and throughput [9].

This trade-off has been referred to by many as "the curse of cursors" and users are often advised by experts about the pitfalls of using cursors [2, 5, 9, 10]. A similar trade-off exists

[1] CURSORs have been a part of ANSI SQL at least since SQL-92.

in database-backed applications where the application code submits a SQL query to fetch data from a remote database and iterates over these query results, performing row-by-row operations. More generally, imperative programs are known to have serious performance problems when they are executed either in a DBMS or in database-backed applications. This area has been seeing more interest recently and there have been several works that have tried to address this problem [19, 23, 24, 26, 36, 38].

In this paper, we present Aggify, a technique for optimizing loops over query results. This loop could either be part of application code that runs on the client, or inside the database as part of UDFs or stored procedures. For such loops, Aggify automatically generates a custom aggregate that is equivalent in semantics to the loop. Then, Aggify rewrites the cursor query to use this new custom aggregate, thereby completely eliminating the loop.

This rewritten form offers the following benefits over the original program. It avoids materialization of the cursor query results and instead, the entire loop is now a single pipelined query execution. It can now leverage existing work on optimization of aggregate functions [21] and result in efficient query plans. In the context of loops in applications that run outside the DBMS, this can significantly reduce the amount of data transferred between the DBMS and the client. Further, the entire loop computation which ran on the client now runs inside the DBMS, closer to data. Finally, all these benefits are achieved without having to perform intrusive changes to user code. As a result, Aggify is a practically feasible approach with many benefits.

The idea that operations in a cursor loop can be captured as a custom aggregate function was initially proposed by Simhadri et. al. [39]. Aggify is based on this principle which applies to any cursor loop that does not modify database state. We formally prove the above result and also show how the limitations given in [39] can be overcome. Aggify can seamlessly integrate with existing works on both optimization of database-backed applications [23, 24] and optimization of UDFs [22, 38]. We believe that Aggify pushes the state of the art in both these (closely related) areas and significantly broadens the scope of prior works. More details can be found in Section 11. Our key contributions are as follows.

(1) We describe Aggify, a language-agnostic technique to optimize loops that iterate over the results of a SQL query. These loops could be either present in applications that run outside the DBMS, or in UDFs/stored procedures that execute inside the DBMS.
(2) We formally characterize the class of loops that can be optimized by Aggify. In particular, we show that Aggify is applicable to all cursor loops present in SQL User-Defined Functions (UDFs). We also prove that the output of Aggify is semantically equivalent to the input cursor loop.
(3) We describe enhancements to the core Aggify technique that expand the scope of Aggify beyond cursor loops to handle iterative FOR loops. We also show how Aggify works in conjunction with existing techniques in this space.
(4) Aggify has been prototyped on Microsoft SQL Server [4]. We discuss the design and implementation of Aggify, and present a detailed experimental evaluation that demonstrates performance gains, resource savings and huge reduction in data movement.

The rest of the paper is organized as follows. We start by motivating the problem in Section 2 and then provide the necessary background in Section 3. Section 4 provides an overview of Aggify and presents the formal characterization. Section 5 and Section 6 describe the core technique, and Section 7 reasons about the correctness of our technique. Section 8 describes enhancements and Section 9 describes the design and implementation. Section 10 presents our experimental evaluation, Section 11 discusses related work, and Section 12 concludes the paper.

2 MOTIVATION

We now provide two motivating examples and then briefly describe how cursor loops are typically evaluated.

2.1 Example: Cursor Loop within a UDF

Consider a query on the TPC-H schema that is based on query 2 of the TPC-H benchmark, but with a slight modification. For each part in the PARTS table, this query lists the part identifier (*p_partkey*) and the name of the supplier that supplies that part with the minimum cost. To this query, we introduce an additional requirement that the user should be able to set a lower bound on the supply cost if required. This lower bound is optional, and if unspecified, should default to a pre-specified value.

Typically, TPC-H query 2 is implemented using a nested subquery. However, another way to implement this is by means of a simple UDF that, given a *p_partkey*, returns the name of the supplier that supplies that part with the minimum cost. Such a query and UDF (expressed in the T-SQL dialect [11]) is given in Figure 1. As described in [38], there are several benefits to implement this as a UDF such as reusability, modularity and readability, which is why developers who are not SQL experts often prefer this implementation.

The UDF *minCostSupp* creates a cursor (in line 6) over a query that performs a join between PARTSUPP and SUPPLIER based on the *p_partkey* attribute. Then, it iterates over the query results while computing the current minimum cost (while ensuring that it is above the lower bound), and

```
--Query:
SELECT p_partkey, minCostSupp(p_partkey) FROM PART

-- UDF definition:
create function minCostSupp(@pkey int, @lb int =-1)
   returns char(25) as
   begin
1     declare @pCost decimal(15,2);
2     declare @minCost decimal(15,2) = 100000;
3     declare @sName char(25), @suppName char(25);

4     if (@lb = -1)
5        set @lb = getLowerBound(@pkey);

6     declare c1 cursor for
      (SELECT ps_supplycost, s_name
         FROM PARTSUPP, SUPPLIER
        WHERE ps_partkey = @pkey
          AND ps_suppkey = s_suppkey);

7     fetch next from c1 into @pCost, @sName;
8     while (@@FETCH_STATUS = 0)
9       if (@pCost < @minCost and @pCost > @lb)
10        set @minCost = @pCost;
11        set @suppName = @sName;
12      fetch next from c1 into @pCost, @sName;
      end
13    return @suppName;
   end
```

Figure 1: Query invoking a UDF that has a cursor loop.

maintaining the name of the supplier who supplies this part at the current minimum (lines 8-12). At the end of the loop, the *@suppName* variable will hold the name of the minimum cost supplier subject to the lower bound constraint, which is then returned from the UDF. Note that for brevity, we have omitted the OPEN, CLOSE and DEALLOCATE statements for the cursor in Figure 1; the complete definition of the loop is available in [13].

This loop is essentially computing a function that can be likened to *argmin*, which is not a built-in aggregate. This example illustrates the fact that cursor loops can contain arbitrary operations which may not always be expressible using built-in aggregates. For the specific cases of functions such as *argmin*, there are advanced SQL techniques that could be used[23]; however, a cursor loop is the preferred choice for developers who are not SQL experts.

2.2 Example: Cursor Loop in a database-backed Application

Consider an application that manages investment portfolios for users. Figure 2 shows a Java method from a database-backed application that uses JDBC API [31] to access a remote database. The table *monthly_investments* includes, among other details, the rate of return on investment (ROI) on a monthly basis. The program first issues a query to retrieve

```
double computeCumulativeReturn(int id, Date from) {
  double cumulativeROI = 1.0;
  Statement stmt = conn.prepareStatement(
      "SELECT roi FROM monthly_investments
       WHERE investor_id = ? and start_date = ?");
  stmt.setInt(1, id);
  stmt.setDate(2, from);

  ResultSet rs = stmt.executeQuery();
  while(rs.next()){
     double monthlyROI = rs.getDouble("roi");
     cumulativeROI =cumulativeROI*(monthlyROI + 1);
  }

  cumulativeROI = cumulativeROI - 1;
  rs.close();   stmt.close();  conn.close();
  return cumulativeROI;
}
```

Figure 2: Java method computing cumulative ROI for investments using JDBC for database access.

all the monthly ROI values for a particular investor starting from a specified date. Then, it iterates over these monthly ROI values and computes the cumulative rate of return on investment using the time-weighted method[2] and returns the cumulative ROI value. Observe that this operation is also not expressible using built-in aggregates.

2.3 Cursor loop Evaluation

A cursor is a control structure that enables traversal over the results of a SQL query. They are similar to iterators in programming languages. DBMSs support different types of cursors such as implicit, explicit, static, dynamic, scrollable, forward-only etc. Our work currently focuses on static, explicit cursors, which are arguably the most widely used.

Cursor loops are usually evaluated as follows. As part of the evaluation of the cursor declaration (the DECLARE CURSOR statement), the DBMS executes the query and materializes the results into a temporary table. The FETCH NEXT statement moves the cursor and assigns values from the current tuple into local variables. The global variable FETCH_STATUS indicates whether there are more tuples remaining in the cursor. The body of the WHILE loop is interpreted statement-by-statement, until FETCH_STATUS indicates that the end of the result set has been reached. Subsequently, the cursor is closed and deallocated in order to clear any temporary work tables created by the cursor.

[2]When the rate of return is calculated over a series of sub-periods of time, the return in each sub-period is based on the investment value at the beginning of the sub-period. Assuming returns are reinvested, if the rates over n successive time sub-periods are $r_1, r_2, r_3, \ldots, r_n$, then the cumulative return rate using the time-weighted method is given by []: $(1+r_1)(1+r_2)\ldots(1+r_n)-1$.

Cursor loops lead to performance issues due to the materialization of the results of the cursor query onto disk, which incurs additional IO and the interpreted evaluation of the loop. This is exacerbated in the presence of large datasets and more so, when invoked repeatedly as in Figure 1. The UDF in Figure 1 is invoked once per part, which means that the cursor query is run multiple times, and temp tables are created and dropped for every run! This is the reason cursors have been referred to as a 'curse' [2, 5, 9, 10].

3 BACKGROUND

We now cover some background material that we make use of in the rest of the paper.

3.1 Custom Aggregate Functions

An aggregate is a function that accepts a collection of values as input and computes a single value by combining the inputs. Some common operations like *min*, *max*, *sum*, *avg* and *count* are provided by DBMSs as built-in aggregate functions. These are often used along with the GROUP BY operator that supplies a grouped set of values as input. Aggregates can be deterministic or non-deterministic. Deterministic aggregates return the same output when called with the same input set, irrespective of the order of the inputs. All the above-mentioned built-in aggregates are deterministic. Oracle's LISTAGG() is an example of a non-deterministic built-in aggregate function [15].

DBMSs allow users to define custom aggregates (also known as User-Defined Aggregates) to implement custom logic. Once defined, they can be used exactly like built-in aggregate functions. These custom aggregates need to adhere to an aggregation contract [1], typically comprising four methods: *init*, *accumulate*, *terminate* and *merge*. The names of these methods may vary across DBMSs. We now briefly describe this contract.

(1) *Init*: Initializes fields that maintain the internal state of the aggregate. It is invoked once per group.
(2) *Accumulate*: Defines the main aggregation logic. It is called once for each qualifying tuple in the group being aggregated. It updates the internal state of the aggregate to reflect the effect of the incoming tuple.
(3) *Terminate*: Returns the final aggregated value. It might optionally perform some computation as well.
(4) *Merge*: This method is optional; it is used in parallel execution of the aggregate to combine partially computed results from different invocations of *Accumulate*.

If the query invoking the aggregate function does not use parallelism, the *Merge* method is never invoked. The other 3 methods are mandatory. The aggregation contract does not enforce any constraint on the order of the input. If order is required, it has to be enforced outside of this contract [15].

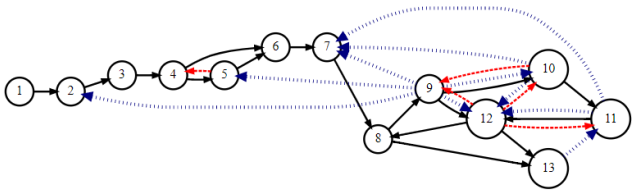


Figure 3: Control Flow Graph for the UDF in Figure 1, augmented with data dependence edges.

Several optimizations on aggregate functions have been explored in previous literature [21]. These involve moving the aggregate around joins and allowing them to be either evaluated eagerly or be delayed depending on cost based decisions [40]. Duplicate insensitivity and null invariance can also be exploited to optimize aggregates [27].

3.2 Data Flow Analysis

We now briefly describe the data structures and static analysis techniques that we make use of in this paper. The material in this section is mainly derived from [17, 32, 33] and we refer the readers to these for further details.

Data flow analysis is a program analysis technique that is used to derive information about the run time behaviour of a program [17, 32, 33]. The *Control Flow Graph (CFG)* of a program is a directed graph where vertices represent basic blocks (a straight line code sequence with no branches) and edges represent transfer of control between basic blocks during execution. The *Data Dependence Graph* (DDG) of a program is a directed multi-graph in which program statements are nodes, and the edges represent data dependencies between statements. Data dependencies could be of different kinds – Flow dependency (read after write), Anti-dependency (write after read), and Output dependency (write after write). The entry and exit point of any node in the CFG is denoted as a *program point*.

Figure 3 shows the CFG for the UDF in Figure 1. Here we consider each statement to be a separate basic block. The CFG has been augmented with data dependence edges where the dotted (blue) and dashed (red) arrows respectively indicate flow and anti dependencies. We use this augmented CFG (sometimes referred to as the Program Dependence Graph or PDG [25]) as the input to our technique.

3.2.1 Framework for data flow analysis. A data-flow value for a program point is an abstraction of the set of all possible program states that can be observed for that point. For a given program entity e, such as a variable or an expression, data flow analysis of a program involves (i) discovering the effect of individual program statements on e (called local data flow analysis), and (ii) relating these effects across statements in the program (called global data flow analysis) by propagating data flow information from one node to another.

The relationship between local and global data flow information is captured by a system of data flow equations. The nodes of the CFG are traversed and these equations are iteratively solved until the system reaches a fixpoint. The results of the analysis can then be used to infer information about the program entity e.

3.2.2 UD and DU Chains. When a variable v is the target of an assignment in a statement S, S is known as a *Definition* of v. When a variable v is on the RHS of an assignment statement S, S is known as a *Use* of v. A Use-Definition (UD) Chain is a data structure that consists of a use U of a variable, and all the definitions D of that variable that can reach that use without any other intervening definitions. A counterpart of a UD Chain is a Definition-Use (DU) Chain which consists of a definition D of a variable and all the uses U, reachable from that definition without any other intervening definitions. These data structures are created using data flow analysis.

3.2.3 Reaching definitions analysis. This analysis is used to determine which definitions reach a particular point in the code [33]. A definition D of a variable reaches a program point p if there exists a path leading from D to p such that D is not overwritten (killed) along the path. The output of this analysis can be used to construct the UD and DU chains which are then used in our transformations. For example, in Figure 1, consider the use of the variable *@lb* inside the loop (line 9). There are at least two definitions of *@lb* that reach this use. One is the the initial assignment of *@lb* to -1 as a default argument, and the other is assignment on line 5.

3.2.4 Live variable analysis. This analysis is used to determine which variables are *live* at each program point. A variable is said to be *live* at a point if it has a subsequent use before a re-definition [33]. For example, consider the variable *@lb* in Figure 1. This variable is *live* at every program point in the loop body. But at the end of the loop, it is no longer *live* as it is never used beyond that point. In the function *minCostSupp*, the only variable that is *live* at the end of the loop is *@suppName*. We will use this information in Aggify as we shall show in Section 5.

4 AGGIFY OVERVIEW

Aggify is a technique that offers a solution to the limitations of cursor loops described in Section 2.3. It achieves this goal by replacing the entire cursor loop with an equivalent SQL query invoking a custom aggregate that is systematically constructed. Performing such a rewrite that guarantees equivalence in semantics is nontrivial. The key challenges involved here are the following. The body of the cursor loop could be arbitrarily complex, with cyclic data dependencies and complex control flow. The query on which the cursor is defined could also be arbitrarily complex, having subqueries, aggregates and so on. Furthermore, the UDF or stored procedure that contains this loop might define variables that are used and modified within the loop.

In the subsequent sections, we show how Aggify achieves this goal such that the rewritten query is semantically equivalent to the cursor loop. Aggify primarily involves two phases. The first phase is to construct a custom aggregate by analyzing the loop (described in Section 5). Then, the next step is to rewrite the cursor query to make use of the custom aggregate and removing the entire loop (described in Section 6).

4.1 Applicability

Before delving into the technique, we formally characterize the class of cursor loops that can be transformed by Aggify and specify the supported operations inside such loops.

Definition 4.1. A *Cursor Loop* (CL) is defined as a tuple (Q, Δ) where Q is any SQL SELECT query and Δ is a program fragment that can be evaluated over the results of Q, one row at a time.

Observe that in the above definition, the body of the loop (Δ) is neither specific to a programming language nor to the execution environment. The loop can be either implemented using procedural extensions of SQL, or using programming languages such as Java. This definition therefore encompasses the loops shown in Figures 1 and 2. In general, statements in the loop can include arbitrary operations that may even mutate the persistent state of the database. Such loops cannot be transformed by Aggify, since aggregates by definition cannot modify database state. We now state the theorem that defines the applicability of Aggify.

Theorem 4.2. *Any cursor loop CL(Q, Δ) that does not modify the persistent state of the database can be equivalently expressed as a query Q' that invokes a custom aggregate function Agg_Δ.*

Proof. We prove this theorem in three steps.

(1) We describe (in Section 5) a technique to systematically construct a custom aggregate function Agg_Δ for a given cursor loop CL(Q, Δ).
(2) We present (in Section 6) the rewrite rule that can be used to rewrite the cursor loop as a query Q' that invokes Agg_Δ
(3) We show (in Section 7) that the rewritten query Q' is semantically equivalent to the cursor loop CL(Q, Δ).

By steps (1), (2), and (3), the theorem follows. □

Observe that Theorem 4.2 encompasses a fairly large class of loops encountered in reality. More specifically, this covers all cursor loops present in user-defined functions (UDFs). This is because UDFs by definition are not allowed to modify the persistent state of the database. As a result, all cursor loops inside such UDFs can be rewritten using Aggify. Note

that this theorem only states that a rewrite is possible; it does not necessarily imply that such a rewrite will always be more efficient. There are several factors that influence the performance improvements due to this rewrite, and we discuss them in our experimental evaluation (Section 10).

4.2 Supported operations

We support all operations inside a loop body that are admissible inside a custom aggregate. The exact set of operations supported inside a custom aggregate varies across DBMSs, but in general, this is a broad set which includes procedural constructs such as variable declarations, assignments, conditional branching, nested loops (cursor and non-cursor) and function invocations. All scalar and table/collection data types are supported. The formal language model that we support is given below.

$$
\begin{aligned}
\mathit{expr} &::= \mathit{Constant} \mid \mathit{var} \mid \textbf{Func(...)} \mid \textbf{Query(...)} \\
&\quad \mid \neg\ \text{expr} \mid \mathit{expr1}\ \textbf{op}\ \mathit{expr2} \\
\mathit{op} &::= + \mid - \mid * \mid / \mid < \mid > \mid \ldots \\
\mathit{Stmt} &::= \mathit{skip} \mid \mathit{Stmt}; \mathit{Stmt} \mid \mathit{var} := \mathit{expr} \\
&\quad \mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt} \\
&\quad \mid \textbf{while}\ \mathit{expr}\ \textbf{do}\ \mathit{Stmt} \\
&\quad \mid \textbf{try}\ \mathit{Stmt}\ \textbf{catch}\ \mathit{Stmt} \\
\mathit{Program} &::= \mathit{Stmt}
\end{aligned}
$$

Nested cursor loops are supported as described in Section 6.3.1. SQL SELECT queries inside the loop are fully supported. DML operations (INSERT, UPDATE, DELETE) on local table variables or temporary tables or collections are supported. Exception handling code (TRY...CATCH) can also be supported. Nested function calls are supported. Operations that may change the persistent state of the database (DML statements against persistent tables, transactions, configuration changes etc.) are not supported. Unconditional jumps such as BREAK and CONTINUE can be supported using boolean variables to keep track of control flow. We can support operations having side-effects only if the DBMS allows these operations inside a custom aggregate. We now describe the core Aggify technique in detail.

5 AGGREGATE CONSTRUCTION

Given a cursor loop (Q, Δ) our goal is to construct a custom aggregate that is equivalent to the body of the loop, Δ. As explained in Section 3.1, we use the aggregate function contract involving the 3 mandatory methods – Init, Accumulate and Terminate – as the target of our construction. Constructing such a custom aggregate involves specifying its signature (return type and parameters), fields and constructing the three method definitions. Figure 4 shows the template that we start with. The patterns <<>> in Figure 4 (shown in green) indicate 'holes' that need to be filled with

```
public class LoopAgg {
   <<  Field declarations for V_F >>

   void Init() { isInitialized = false; }
   void Accumulate(<< Param specs for P_accum >>) {
      if (!isInitialized) {
         << Assignments for V_init >>
         isInitialized = true;
      }
      << Loop body Δ >>
   }
   <<TYPE(V_term)>> Terminate(){ return << V_term >>; }
}
```

Figure 4: Template for the custom aggregate.

```
public class MinCostSuppAgg {
   double minCost; string suppName;
   int lb; bool isInitialized;

   void Init() { isInitialized = false; }

   void Accumulate(double pCost, string sName,
                   double pMinCost, int pLb) {
      if (!isInitialized) {
         minCost = pMinCost;
         lb = pLb;
         isInitialized = true;
      }
      if (pCost < minCost && pCost > lb) {
         minCost = pCost;
         suppName = sName;
      }
   }
   string Terminate() { return suppName; }
}
```

Figure 5: Custom aggregate for the loop in Figure 1.

```
public class CumulativeReturnAgg {
   double cumulativeROI; bool isInitialized;

   void Init() { isInitialized = false; }

   void Accumulate(double monthlyROI,
                   double pCumulativeROI) {
      if (!isInitialized) {
         cumulativeROI = pCumulativeROI;
         isInitialized = true;
      }
      cumulativeROI=cumulativeROI*(monthlyROI + 1);
   }
   double Terminate() { return cumulativeROI; }
}
```

Figure 6: Custom aggregate for the loop in Figure 2.

code fragments inferred from the loop. We now show how to construct such an aggregate and illustrate it using the examples from Section 2. Figures 5 and 6 show the definition of the custom aggregate for the loops in Figures 1 and 2 respectively. We use the syntax of Microsoft SQL Server to

illustrate these examples; however the technique applies to other SQL dialects as well.

5.1 Fields

Conservatively, all variables live at the beginning of the loop can be made fields of the aggregate. We identify a minimal set of fields as follows. Consider the set V_Δ of all variables referenced in the loop body Δ. Let V_{fetch} be the set of variables assigned in the *FETCH* statement, and let V_{local} be the set of variables that are local to the loop body (i.e they are declared within the loop body and are not *live* at the end of the loop.) The set of variables V_F defined as fields of the custom aggregate is given by the equation:

$$V_F = (V_\Delta - (V_{fetch} \cup V_{local})) \cup \{isInitialized\}. \quad (1)$$

We have additionally added a variable called *isInitialized* to the field variables set V_F. This boolean field is necessary for keeping track of field initialization, and will be described in Section 5.2. For all variables in V_F, we place a field declaration statement in the custom aggregate class.

Illustrations: For the loop in Figure 1,

$$V_\Delta = \{pCost, minCost, lb, suppName, sName\}$$
$$V_{fetch} = \{pCost, sName\}$$
$$V_{local} = \{\}$$

Therefore, using Equation 1, we get

$$V_F = \{minCost, lb, suppName, isInitialized\}\square$$

For application programs such as the one in Figure 2 that use a data access API like JDBC, the attribute accessor methods (e.g. *getInt(), getString()* etc.) on the *ResultSet* object are treated analogous to the *FETCH* statement. Therefore, local variables to which *ResultSet* attributes are assigned form a part of the V_{fetch} set. For the loop in Figure 2,

$$V_\Delta = \{cumulativeROI, monthlyROI\}$$
$$V_{fetch} = \{monthlyROI\}$$
$$V_{local} = \{\}$$

Therefore, using Equation 1, we get

$$V_F = \{cumulativeROI, isInitialized\}\square$$

5.2 Init()

The implementation of the *Init()* method is very simple. We just add a statement that assigns the boolean field *isInitialized* to *false*. Initialization of field variables is deferred to the *Accumulate()* method for the following reason. The *Init()* does not accept any arguments. Hence if field initialization statements are placed in *Init()*, they will have to be restricted to values that are statically determinable [39]. This is because these values will have to be supplied at aggregate function creation time. In practice it is quite likely that these values are not statically determinable. This could be because (a) they are not compile-time constants but are variables that hold a value at runtime, or (b) there are multiple definitions of these variables that might reach the loop, due to presence of conditional assignments.

Consider the loop of Figure 1. Based on Equation 1, we have determined that the variable *@lb* has to be a field of the custom aggregate. Now, we cannot place the initialization of *@lb* in *Init()* because there is no way to determine the initial value of *@lb* at compile-time using static analysis of the code. This was a restriction in [39] which we overcome by deferring field initializations to *Accumulate()*.

Illustrations: The *Init()* method is identical in both Figures 5 and 6, having an assignment of *isInitialized* to *false*.

5.3 Accumulate()

In a custom aggregate, the *Accumulate()* method encapsulates the important computations that need to happen. We now construct the parameters and the definition of *Accumulate()*.

5.3.1 Parameters. Let P_{accum} denote the set of parameters which is identified as *the set of variables that are used inside the loop body and have at least one reaching definition outside the loop*. The set of candidate variables is computed using the results of reaching definitions analysis (Section 3.2.3). More formally, let V_{use} be the set of all variables used inside the loop body. For each variable $v \in V_{use}$, let $U_{CL}(v)$ be the set of all uses of v inside the cursor loop CL. Now, for each use $u \in U_{CL}(v)$, let RD(u) be the set of all definitions of v that reach the use u. We define a function $R(v)$ as follows.

$$R(v) = \begin{cases} 1, & \text{if } \exists d \in RD(u) \mid d \text{ is not in the loop.} \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Checking if a definition d is in the loop or not is a simple set containment check. Using Equation 2, we define P_{accum}, the set of parameters for *Accumulate()* as follows.

$$P_{accum} = \{v \mid v \in V_{use} \wedge R(v) == 1\} \quad (3)$$

5.3.2 Method Definition. There are two blocks of code that form the definition of *Accumulate()* – field initializations and the loop body block. The set of fields V_{init} that need to be initialized is given by the below equation.

$$V_{init} = P_{accum} - V_{fetch} \quad (4)$$

As mentioned earlier, the boolean field *isInitialized* denotes whether the fields of this class are initialized or not. The first time accumulate is invoked for a group, *isInitialized* is false and hence the fields in V_{init} are initialized. During subsequent invocations, this block is skipped as *isInitialized* would be true. Following the initialization block, the entire loop body Δ is appended to the definition of *Accumulate()*.

```
create function minCostSupp(@pkey int, @lb int =-1)
returns char(25) as
begin
   declare @minCost decimal(15,2) = 100000;
   declare @suppName char(25);

   if (@lb = -1)
      set @lb = getLowerBound(@pkey);

   set @suppName = (
       SELECT MinCostSuppAgg(Q.ps_supplycost,
                             Q.s_name, @minCost, @lb)
         FROM (SELECT ps_supplycost, s_name
                 FROM PARTSUPP, SUPPLIER
                WHERE ps_partkey = @pkey
                  AND ps_suppkey = s_suppkey) Q );
    return @suppName;
end
```

Figure 7: The UDF in Figure 1 rewritten using Aggify.

Illustrations: For the loop in Figure 1:

$$P_{accum} = \{pCost, sName, pMinCost, pLb\}$$
$$V_{init} = \{minCost, lb\}$$

For the loop in Figure 2, P_{accum} and V_{init} are as follows:

$$P_{accum} = \{monthlyROI, cumulativeROI\}$$
$$V_{init} = \{cumulativeROI\}$$

The *Accumulate()* method in Figures 5 and 6 are constructed based on the above equations as per the template in Figure 4.

5.4 Terminate()

This method returns a tuple of all the field variables (V_F) that are *live* at the end of the loop. The set of candidate variables V_{term} are identified by performing a liveness analysis for the module enclosing the cursor loop (e.g. the UDF that contains the loop). The return type of the aggregate is a tuple where each attribute corresponds to a variable that is live at the end of the loop. The tuple datatype can be implemented using User-Defined Types in most DBMSs.

Illustrations: For the loop in Figure 1, $V_{term} = \{suppName\}$, and for the loop in Figure 2, $V_{term} = \{cumulativeROI\}$. For simplicity, since these are single-attribute tuples, we avoid using a tuple and use the type of the attribute as the return type of *Terminate()*.

6 QUERY REWRITING

For a given cursor loop (Q, Δ), once the custom aggregate Agg_Δ has been created, the next task is to remove the loop altogether and rewrite the query Q into Q' such that it invokes this custom aggregate instead. Note that Q might be arbitrarily complex, and may contain other aggregates (built-in or custom), GROUP BY, sub-queries and so on. Therefore,

```
double computeCumulativeReturn(int id, Date from) {
  double cumulativeROI = 1.0;

  Statement stmt = conn.prepareStatement(
     "SELECT CumulativeReturnAgg(Q.roi, ?) AS croi
       FROM (SELECT roi FROM monthly_investments
              WHERE investor_id = ?
                AND start_date = ?) Q");

  stmt.setDouble(1, cumulativeROI);
  stmt.setInt(2, id);  stmt.setDate(3, from);

  ResultSet rs = stmt.executeQuery();
  rs.next();
  cumulativeROI = rs.getDouble("croi");

  cumulativeROI = cumulativeROI - 1;
  rs.close();   stmt.close();  conn.close();
  return cumulativeROI;
}
```

Figure 8: The Java method in Figure 2 rewritten using Aggify.

Aggify constructs Q' without modifying Q directly, but by composing Q as a nested sub-query. In other words, Aggify introduces an aggregation on top of Q that contains an invocation to Agg_Δ. Note that Agg_Δ is the only attribute that needs to be projected, as it contains all the loop variables that are *live*. In relational algebra, this rewrite rule can be represented as follows:

$$\text{Loop}(Q, \Delta) \implies \mathcal{G}_{\text{Agg}_\Delta(P_{accum}) \text{ as aggVal}}(Q) \quad (5)$$

Note that the parameters to Agg_Δ are the same as the parameters to the *Accumulate()* method (P_{accum}). These are either attributes that are projected from Q or variables that are defined earlier. The return value of Agg_Δ (aliased as *aggVal*) is a tuple from which individual attributes can be extracted. The details are specific to SQL dialects.

Illustration: Figure 7 shows the output of rewriting the UDF in Figure 1 using Aggify. Observe the statement that assigns to the variable *@suppName* where the R.H.S is a SQL query. This is the resulting query corresponding to the loop in Figure 1; the aggregate *MinCostSuppAgg* is defined in Figure 5.

Figure 8 shows the Java method from Figure 2 rewritten using Aggify. Out of the 2 parameters to the aggregate function, one is an attribute from the underlying query, and the other is a local variable. The loop is replaced with a method that advances the *ResultSet* to the first (and only) row in the result, and an attribute accessor method invocation (*getDouble()* in this case) is placed with an assignment to each of the *live* variables (*cumulativeROI* in this case).

6.1 Order enforcement

The query Q over which a cursor is defined may be arbitrarily complex. If Q does not have an ORDER BY clause, the DBMS gives no guarantee about the order in which the rows are iterated over. Equation 5 is in accordance with this, because the DBMS gives no guarantee about the order in which the custom aggregate is invoked as well. Hence the above query rewrite suffices in this case.

However, the presence of ORDER BY in the cursor query Q implies that the loop body Δ is invoked in a specific order determined by the sort attributes of Q. In this case, the above rewriting is not sufficient as it does not preserve the ordering and may lead to wrong results. Therefore, Simhadri et. al [39] mention that either there should be no ORDER BY clause in the cursor query, or the database system should allow order enforcement while invoking custom aggregates. To address this, we now propose a minor variation of the above rewrite rule that can be used to enforce the necessary order.

Let Q_s represent a query with an ORDER BY clause where the subscript s denotes the sort attributes. Let Q represent the query Q_s without the ORDER BY clause. For a cursor loop (Q_s, Δ), the rewrite rule can be stated as follows:

$$\text{Loop}(Q_s, \Delta) \implies \mathcal{G}_{\text{StreamAgg}_\Delta(P_{accum}) \text{ as aggVal}}(Sort_s(Q)) \quad (6)$$

This rule enforces two conditions. (i) It enforces the sort operation to be performed before the aggregate is invoked, and (ii) it enforces the *Streaming Aggregate* physical operator to implement the custom aggregate. These two conditions ensure that the order specified in the cursor loop is respected.

6.2 Discussion

Once the query is rewritten as described above, Aggify replaces the loop with an invocation to the rewritten query as shown in Figures 7 and 8. The return value of the aggregate is assigned to corresponding local variables, which enables subsequent lines of code to remain unmodified. From Figures 7 and 8, we can make the following observations.

- The cursor query Q remains unchanged, and is now the subquery that appears in the FROM clause.
- The transformation is fairly non-intrusive. Apart from the removal of the loop, the rest of the lines of code remain identical, except for a few minor modifications.
- This transformation may render some variables as *dead*. Declarations of such variables can be then removed, thereby further simplifying the code [33]. For instance, the variables *@pCost* and *@sName* in Figure 1 are no longer required, and are removed in Figure 7.

The transformed program Aggify offers the following benefits. It avoids materialization of the cursor query results and instead, the entire loop is now a single pipelined query execution. In the context of loops in applications that run outside the DBMS (Figure 8), this rewrite reduces the amount of data transferred between the DBMS and the client. Further, the entire loop computation now runs inside the DBMS, closer to data. Finally, all these benefits are achieved without intrusive changes to source code.

6.3 Aggify Algorithm

Algorithm 1 Aggify(G, Q, Δ)

Require: G: CFG of the program augmented with data dependence edges;
Q: Cursor query;
Δ: Subgraph of G for the loop body;

$A(L, R, UD, DU) \leftarrow$ Perform DataFlow Analysis on G;
  $L \leftarrow$ Liveness information;
  $RD \leftarrow$ Reachable Definitions;
  $UD, DU \leftarrow$ Use-Def Chain, Def-Use Chain;
$V_\Delta \leftarrow$ {Variables referenced in Δ};
$V_{fetch} \leftarrow$ {Vars. assigned in the FETCH statement};
$V_{field} \leftarrow$ {Compute using Equation 1};
$P_{accum} \leftarrow$ {Compute using Equation 3};
$V_{init} \leftarrow$ {Compute using Equation 4};
$V_{term} \leftarrow$ {Fields that are live at loop end};

$Agg_\Delta \leftarrow$ Construct aggregate class using template in Figure 4 and above information;
Register Agg_Δ with the database;

if (Q contains ORDER BY clause) **then**
  $s \leftarrow$ {ORDER BY attributes}
  Rewrite loop using Equation 6;
else
  Rewrite loop using Equation 5;

The entire algorithm illustrated in Sections 5 and 6 is formally presented in Algorithm 1. The algorithm accepts G, the CFG of the program augmented with data dependence edges; Q, the cursor query; and Δ, the subgraph of G corresponding to the loop body. Algorithm 1 is invoked for every loop after necessary preconditions in Section 4.2 are satisfied.

Initially, we perform DataFlow Analyses on G as described in Section 3. The results of these analyses are captured as $A(L, RD, UD, DU)$ which consists of Liveness, Reachable definitions, Use-Def and Def-Use chains respectively. Then, these results are used to compute the necessary components for the aggregate definition, namely V_Δ, V_{init}, V_{fetch}, V_{field}, V_{term}, P_{accum}. Once all the necessary information is computed, the aggregate definition is constructed using the template in Figure 4, and this aggregate (called Agg_Δ) is registered with the database engine. Finally, we rewrite the entire loop with

a query that invokes Agg_Δ. The rewrite rule is chosen based on whether the cursor query Q has an ORDER BY clause, as described in Section 6.

6.3.1 Nested cursor loops: Cursors can be nested, and our algorithm can handle such cases as well. This can be achieved by first running Algorithm 1 on the inner cursor loop and transforming it into a SQL query. Subsequently, we can run Algorithm 1 on the outer loop. An example is provided (L8-W2) in customer workload experiments in [13].

7 PRESERVING SEMANTICS

We now reason about the correctness of the transformation performed by Aggify, and describe how the semantics of the cursor loop are preserved.

Let CL(Q, Δ) be a cursor loop, and let Q' be the rewritten query that invokes the custom aggregate. The program state comprises of values for all *live* variables at a particular program point. Let P_0 denote the program state at the beginning of the loop and P_n denote the program state at the end of the loop, where $n = |Q|$. To ensure correctness, we must show that if the execution of the cursor loop on P_0 results in P_n, then the execution of Q' on P_0 also results in P_n. We only consider program state and not the database state in this discussion, as our transformation only applies to loops that do not modify the database state.

Every iteration of the loop can be modeled as a function that transforms the intermediate program state. Formally,

$$P_i = f(P_{i-1}, T_i)$$

where i ranges from 1 to n. In fact the function f would be comprised of the operations in the loop body Δ.

It is now straightforward to see that the *Accumulate()* method of the custom aggregate constructed by Aggify exactly mimics this behavior. This is because (a) the statements in the loop body Δ are directly placed in the *Accumulate()* method, (b) the *Accumulate()* is called for every tuple in Q, and (c) the rule in Equation 6 ensures that the order of invocation of *Accumulate()* is identical to that of the loop when necessary. The fields of the aggregate class[3] and their initialization ensure the inter-iteration program states are maintained. From our definition of V_{term} in Section 5.4, it follows that $P_n = V_{term}$. Therefore the output of the custom aggregate is identical to the program state at the end of the cursor loop. □

8 ENHANCEMENTS

We now present enhancements to Aggify that further broaden its applicability.

[3]Note that here $V_F = P_0$. In other words, we consider all variables that are live at the beginning of the loop (i.e. P_0) as fields of the aggregate. This is a conservative but correct definition as given in Section 5.1.

8.1 Optimizing Iterative FOR Loops

Although the focus of Aggify has been to optimize loops over query results, the technique can be extended to more general FOR loops with a fixed iteration space. A FOR loop is a control structure used to write a loop that needs to execute a specific number of times. Such loops are extremely common, and typically have the following structure:

```
FOR (init; condition; increment) { statement(s);}
```

Such loops can be written as cursor loops by expressing the iteration space as a relation. Consider this loop

```
FOR (i = 0; i <= 100; i++) { statement(s);}
```

The iteration space of this loop can be written as a SQL query using either recursive CTEs, or vendor specific constructs (such as DUAL in Oracle). The above loop written using a recursive CTE is given below.

```
with CTE as (  select 0 as i
  union all select i + 1  from CTE where i <= 100
)  select * from CTE
```

Now, we can define a cursor on the above query with the same loop body. This is now a standard cursor loop and hence Aggify can be used to optimize it. Rewriting FOR loops as recursive CTEs can be achieved by extracting the *init*, *condition* and *increment* expressions from the FOR loop, and placing them in the CTE template given above. We omit details of this transformation.

8.2 Extending existing techniques

We now show how Aggify can seamlessly integrate with existing optimization techniques, both in the case of applications that run outside the DBMS and UDFs that run within.

There have been recent efforts to optimize database applications using techniques from programming languages [19, 23, 24, 37]. In fact, [23] mentions that loops over query results could be converted to user-defined aggregates, but do not describe a technique for the same. Aggify can be used as a preprocessing step which can replace loops with equivalent queries which invoke a custom aggregate. Then, the techniques of [23] can be used to further optimize the program.

The Froid framework [38], which was also based on the work of Simhadri et. al [39] showed how to transform UDFs into sub-queries that could then be optimized. However, Froid cannot optimize UDFs with loops. Building upon this idea, Duta et. al [22] described a technique to transform arbitrary loops into recursive CTEs. While this avoids function call overheads and context switching, it is limited by the optimization techniques that currently exist for recursive CTEs. For the specific case of cursor loops, Aggify avoids creating recursive CTEs.

These ideas can be used together in the following manner: (i) If the function has a cursor loop, use Aggify to eliminate it.

(ii) If the function has a FOR loop and the necessary expressions can be extracted from it, use the technique described in Section 8.1 along with Aggify to eliminate the loop. (iii) If the function has an arbitrary loop with a dynamic iteration space, use the technique of Duta et. al [22]. After applying (i), (ii), or (iii), Froid can be used to optimize the resulting loop-free UDF.

9 IMPLEMENTATION

The techniques described in this paper can be implemented either inside a DBMS or as an external tool. We have currently implemented a prototype of Aggify in Microsoft SQL Server. Aggify currently supports cursor loops that are written in Transact-SQL [11], and constructs a user-defined aggregate in C# [20]. Note that translating from T-SQL into C# might lead to loss of precision and sometimes different results due to difference in data type semantics. We are currently working on a better approach which is to natively implement this inside the database engine [28].

Implementing Aggify inside a DBMS allows the construction of more efficient implementations of custom-aggregates that can be baked into the DBMS itself. Also, observe that the rule in Equation 6 that enforces streaming aggregate operator for the custom aggregate has to be part of the query optimizer. In fact, apart from this rule, there is no other change required to be made to the query optimizer. Every other part of Aggify can be implemented outside the query optimizer. However, we note that since Microsoft SQL Server only supports the Streaming Aggregate operator for user-defined aggregates, we did not have to implement Equation 6.

Froid [38] is available as a feature called *Scalar UDF Inlining* [8] in Microsoft SQL Server 2019. As mentioned in Section 8.2, Aggify integrates seamlessly with Froid, thereby extending the scope of Froid to also handle UDFs with cursor loops. Aggify is first used to replace cursor loops with equivalent SQL queries with a custom aggregate; this is then followed by Froid which can now inline the UDF.

10 EVALUATION

We now present some results of our evaluation of Aggify. Our experimental setup is as follows. Microsoft SQL Server 2019 [4] with Aggify was run on Windows 10. The machine was equipped with an Intel Quad Core i7, 3.6 GHz, 64 GB RAM, and SSD-backed storage. The SQL cursor loops were run directly on this machine, and did not involve any network. The client applications were run on a remote machine connected to the DBMS over LAN.

10.1 Workloads

We have evaluated Aggify on many workloads on several data sizes and configurations. We show results based on 3 real workloads, an open benchmark based on TPC-H queries, and several Java programs including an open benchmark. All these queries, UDFs and cursor loops are made available [13].

TPC-H Cursor Loop workload: To evaluate Aggify on an open benchmark that mimics real workloads, we implemented the specifications of a few TPC-H queries using cursor loops. Not all TPC-H queries are amenable to be written using cursor loops, so we have chosen a logically meaningful subset. While this is a synthetic benchmark, it illustrates common scenarios found in real workloads. We report results on the 10GB scale factor. The database had indexes on L_ORDERKEY and L_SUPPKEY columns of LINEITEM, O_CUSTKEY column of ORDERS, and PS_PARTKEY column of PARTSUPP. For this workload we show a breakdown of results for Aggify, and Aggify+ (Froid applied after Aggify).

Real workloads: We have considered 3 real workloads (proprietary) for our experiments. While we have access to the queries and UDFs/stored procedures of these workloads, we did not have access to the data. As a result, we have synthetically generated datasets that suit these workloads. We have also manually extracted required program fragments from these workloads so that we can use them as inputs to Aggify. Workload W1 is a CRM application, W2 is a configuration management tool, and W3 is a backend application for a transportation services company. Note that we have not combined Aggify with Froid in these workloads, as we did not have access to the queries that invoke these UDFs.

Java workload: The implementation of Aggify for Java is ongoing. For performance evaluation, we have considered the RUBiS benchmark [7] and two other examples and manually transformed them using Algorithm 1. One of them is a Java implementation of the minimum cost supplier functionality similar to the example in Figure 1. The other is a variant of the example in Figure 2 with 50 columns. These programs are also available in [13]. Note that Froid is applicable only to T-SQL, so the Java experiments do not use Froid.

10.2 Applicability of Aggify

We have analyzed several real world workloads and open-source benchmark applications to measure (a) the usage of cursor loops, and (b) the applicability of Aggify on such loops. We considered about 5720 databases in Azure SQL Database [14] that make use of UDFs in their workload. Across these databases, we came across more than 77,294 cursors being declared inside UDFs[4]. As explained in Section 4.1, Aggify can be used to rewrite all these cursor loops. This demonstrates both the wide usage of cursor loops in real workloads, and the broad applicability of Aggify.

[4]This analysis was done using scripts that analyze the database metadata and extract the necessary information. We do not have access to manually examine the source code of these propreitary UDFs.

Table 1: Applicability of Aggify

Workload	RUBiS	RUBBoS	Adempiere
Total # of while loops	16	41	127
# of cursor loops	14 (87.5%)	14 (34.14%)	109 (85.8%)
Aggify-able	14	14	>80

Next, we manually analyzed 3 opensource Java applications – the RUBiS Benchmark [7], RUBBos Benchmark [6], and the popular Adempiere [12] CRM application. Table 1 shows the results of this analysis. 87.5% of the loops in the RUBiS benchmark were cursor loops. In RUBBoS, 34% of the loops were cursor loops. We looked at a subset of files in Adempiere ($\tilde{2}5$ files) where more than 85% of the loops encountered were cursor loops. Interestingly all the cursor loops in RUBiS and RuBBoS, and more than 80 loops in Adempiere satisfied the preconditions for Aggify. This shows the use of cursors as well as the applicability of Aggify. More details of this analysis can be found in [28].

10.3 Performance improvements

We now show the results of our experiments to determine the overall performance gains achieved due to Aggify.

10.3.1 TPC-H workload. First, we consider the TPC-H cursor loop workload and show the results on a 10 GB database with warm buffer pool. Similar trends have been observed with 100GB as well, with both warm and cold buffer pool configurations. Figure 9(a) shows the results for 6 queries from the workload [13]. The solid column (in blue) represents the original program. The striped column (orange) represents results of applying Aggify. The green column (indicated as 'Aggify+' shows the results of applying Froid's technique after Aggify enables it, as described in Section 8.2.

On the x-axis, we indicate the query number, and on the y-axis, we show execution time in seconds, in log scale. Observe that for queries Q2, Q13 and Q21, we have a ⊘ symbol above the column corresponding to the original query with the loop, and for Q13, we have that symbol even for the Aggify column. This means that we had to forcibly terminate these queries as they were running for a very long time (>10 days for Q2, >22 days hours for Q13 and >9 hours for Q21). We observe that Q2, Q14, Q18 and Q21 offer at least an order of magnitude improvement purely due to Aggify alone. When Aggify is combined with Froid, we see further improvements in Q2, Q13, Q18 and Q19. Q13 results in a huge improvement of 3 orders of magnitude due to the combination. Note that without Aggify, Froid will not be able to rewrite these queries at all. Q21 does not lead to any additional gains from Froid, while Q14 slows down slightly due to Froid.

10.3.2 Java workload. Figure 9(b) shows the results of running Aggify on 5 loops from the RUBis [7] benchmark. The x-axis indicates the 5 scenarios along with the number of iterations of the loop (given in parenthesis), and y-axis shows the execution time. As before, the blue column indicates the original program, and the dashed orange column indicates the results with Aggify. We observe that Aggify improves performance for all these scenarios. Here the beneifts due to Aggify stem mainly from the huge reduction in data transfer between the database server and the client Java application.

Table 2: Comparison of logical reads for the TPC-H cursor loop benchmark.

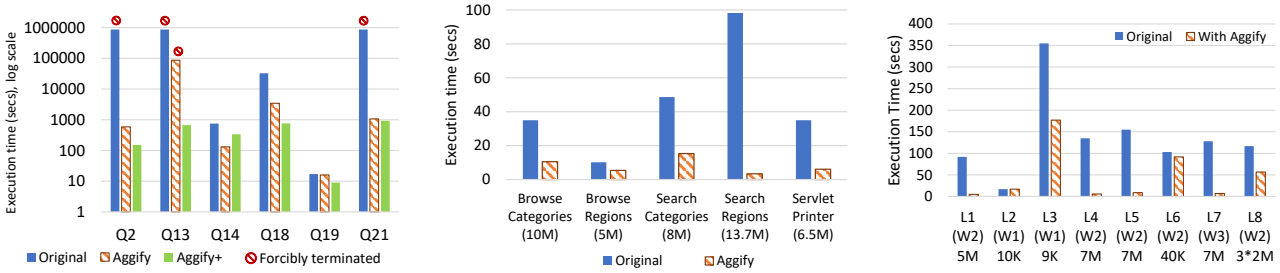
Qry	Original	Aggify	Aggify+	Savings w.r.t Original	
				Aggify	Aggify+
Q2	⊘	38.1M	54.4M	NA	NA
Q13	⊘	⊘	0.26M	NA	NA
Q14	553M	11.7M	231M	541M	322M
Q18	405M	120M	293M	285M	113M
Q19	1.11M	1.11M	1.11M	4528	4611
Q21	⊘	464M	616M	NA	NA

10.3.3 Real workloads. Now, we consider loops that we encountered in customer workloads W1, W2 and W3 and run them with and without Aggify. Figure 9(c) shows the results of using Aggify on 8 of these loops. The y-axis shows execution time in seconds for loops L1-L8. The iteration counts are given on the x-axis labels. We observe improvements in most cases, ranging from 2x to 22x. Note that loop L8 in Figure 9(c) is a nested cursor loop that gives more than 2x gains. Loops L2 and L6 iterate over a relatively small number of tuples compared to the others. This is one cause for the small or no performance gains. Also, these two loops included many statements that inserted values into temporary tables or table variables. For such statements, in our implementation of Aggify, we have to make a connection to the database explicitly in order to insert these tuples. That adds additional overhead, which could be avoided if the aggregate is implemented natively inside the DBMS (Section 9).

10.4 Resource Savings

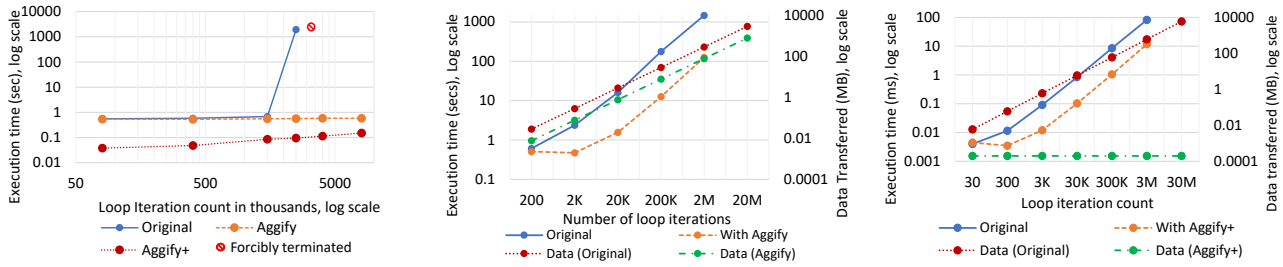
In addition to performance gains, Aggify also reduces resource consumption, primarily disk IO. This is because cursors end up materializing query results to disk, and then reading from the disk during iteration, whereas the entire loop runs in a pipelined manner with Aggify. To illustrate this, we measured the logical reads incurred on our workloads. Table 2 shows these numbers (in millions) for 6 queries from the TPC-H cursor loop benchmark. For the original programs, we have the numbers for 3 queries as we had to forcibly terminate the others as mentioned in Section 10.3.

We see that Aggify significantly brings down the required number of reads. Q14 and Q18 show huge reductions (58% and 27% respectively). Table 2 shows the breakup of logical



(a) TPC-H cursor loop workload. **(b) RUBiS Benchmark (Java)** **(c) Customer workloads.**

Figure 9: Performance improvements across multiple workloads



(a) TPC-H Q2 (MinCostSupplier). **(b) MinCostSupplier (Java)** **(c) CumulativeROI (Java).**

Figure 10: Scalability across multiple workloads (varying loop iteration counts).

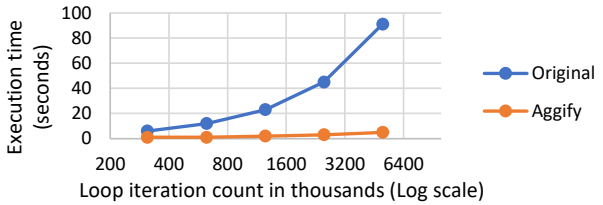


Figure 11: Scalability for loop L1 (workload W2).

reads with *Aggify* alone (column 3), and *Aggify+* (column 4) which denotes Aggify with Froid. Columns 5 and 6 denote the savings in logical reads due to Aggify and Aggify+. Interestingly, we observe that using Froid with aggify results in more logical reads, but improves execution time.

10.5 Scalability

We now show the results of our experiments to evaluate the scalability of Aggify with varying data sizes. Figures 10(a), (b), (c) and Figure 11 show the results for experiments described below. The x-axis shows the number of loop iterations, and y-axis shows execution time in all 4 experiments.

Experiment 1: Figure 10(a) shows the results for TPC-H query Q2. We show the results for the original UDF (blue), the UDF transformed using Aggify (orange), and further applying Froid (red) indicated as *Aggify+*. For smaller sizes, Aggify does not offer any improvement by itself. Beyond a certain point, the original program degrades drastically, while Aggify stays constant. For *Aggify+*, we observe about an order of magnitude improvement in performance all through.

Experiment 2: Next, we consider the Java implementation of minimum cost supplier functionality. The original program first runs a query that retrieves the required number of parts, and then the loop iterates over these parts, and computes the minimum cost supplier for each. We restrict the number of parts using a predicate on P_PARTKEY. By varying its value, we can control the iteration count of the loop. There were 2 million tuples in the PART table, and hence we vary the iteration count from 200 to 2 million in multiples of 10. The transformed Java program eliminates this loop completely, and executes a query that makes use of the custom aggregate *MinCostSuppAgg*.

Figure 10(b) shows the results of this experiment with warm cache. The solid line (in blue) and the dashed line (in orange) represent the original program and the transformed program respectively. We observe that at smaller number of iterations, the benefits are lesser, but beyond 2K iterations, we see a consistent improvement by an order of magnitude.

Experiment 3: We consider a variant of the example described in Section 2.2 (Figure 2) that computes the cumulative rate of return on investments. The table had 50 columns

which store the monthly rate of return per investment category for that investor. The TOP keyword is used to control the iteration count of the loop from 30 to 3 million in multiples of 10. Figure 10(c) shows the results of this experiment. We see that beyond 3K iterations, Aggify starts to offer an order of magnitude improvement, all the way up to 3 million.

This transformation uses Aggify to eliminate the loop, and then follows the technique of [23] as described in Section 8.2. Without Aggify, the technique of [23] will not be able to translate this loop into SQL. The benefits for these two experiments are due to a combination of (i) pushing compute from the remote Java application into the DBMS, (ii) reducing the amount of data transferred from the DBMS to the application, and (iii) the SQL translation of [23].

Experiment 4: We consider loop L1 from the real workload W1 (the loop is given in [13]) and vary loop iteration count; the results are given in Figure 11. The benefits of Aggify get better with scale, similar to the other scalability experiments. These benefits arise due to pipelining as well as reduction in data movement.

10.6 Data Movement

One of the key benefits due to Aggify is the reduction in data movement from a remote DBMS to client applications. We measure the magnitude of data moved, and show how Aggify significantly reduces this. The results of this experiment for the MinCostSupplier and Cumulative ROI Java programs are plotted in Figures 10(b) and 10(c) using the secondary y-axis. In both figures, the dotted line (red) shows the data moved from the DBMS to the client for the original program in megabytes, and the dash-dot line (green) shows the data movement for the rewritten program.

For the MinCostSupplier experiment, the original program ends up transferring $(140 * n)$ bytes of data where n is the number of iterations (i.e. number of parts), assuming 4-byte integers (P_PARTKEY), 9-byte decimals(PS_SUPPLYCOST) and 25-byte varchars (S_NAME). The rewritten program transfers only $(38 * n)$ bytes, resulting in a reduction of 3.6x. For the CumulativeROI experiment, the original program transfers 200 bytes per iteration (assuming 4-byte floating point values). At 30 million tuples, this is 6GB of data transfer! Aggify only returns the result of the computation, a single tuple with 50 floating point values (200 bytes) irrespective of the number of iterations.

11 RELATED WORK

Optimization of loops in general, has been an active area of research in the compilers/PL community. Techniques for loop parallelization, tiling, fission, unrolling etc. are mature, and are part of state-of-the-art compilers [32, 35]. Lieuwen and DeWitt [34] describe techniques to optimize set iteration loops in object oriented database systems (OODBs). They show how to extend compilers to include database-style optimizations such as join reordering.

There have been recent works that have explored the use of program synthesis to address problems such as (a) optimization of applications that use ORMs [19], (b) translation of imperative programs into the Map Reduce paradigm [16, 37]. In contrast to these works, Aggify relies on program analysis and query rewriting. Further, Aggify expresses an entire loop as a relational aggregation operator. Cheung et al. [18] show how to partition database application code such that part of the code runs inside the DBMS as a stored procedure. Aggify also pushes computation into the DBMS, but moves entire cursor loops as an aggregate function thereby leveraging optimization techniques for aggregate functions [21].

The idea of expressing loops as custom aggregates was first proposed by Simhadri et. al. [39] as part of the UDF decorrelation technique. Aggify is based on this idea. We (i) formally characterize the class of cursor loops that can be transformed into custom aggregates, (ii) relax a pre-condition given in [39] thereby expanding the applicability of this technique, and (iii) show how this technique extends to FOR loops and applications that run outside the DBMS.

The DBridge line of work [23, 24, 29] has had many contributions in the area of optimizing data access in database applications using static analysis and query rewriting. [29] consider the problem of rewriting loops to make use of parameter batching. Emani et. al [23] describe a technique to translate imperative code to equivalent SQL. Recently, there have been efforts to optimize UDFs by transforming them into sub-queries or recursive CTEs [22, 38]. As we have shown in Section 8.2, Aggify can be used in conjunction with all these techniques leading to better performance.

12 CONCLUSION

Although it is well-known that set-oriented operations are generally more efficient compared to row-by-row operations, there are several scenarios where cursor loops are preferred, or are even inevitable. However, due to many reasons that we detail in this paper, cursor loops can not only result in poor performance, but also affect concurrency and resource consumption. Aggify, the technique presented in this paper, addresses this problem by automatically replacing cursor loops with SQL queries that invoke custom aggregates that are automatically constructed based on the loop body. Our evaluation on benchmarks and real workloads show the potential benefits of such a technique. We believe that Aggify, can positively impact real-world workloads both in database-backed applications as well as UDFs and stored procedures.

REFERENCES

[1] [n.d.]. CLR User-Defined Aggregates - Requirements. https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-user-defined-aggregates-requirements?view=sql-server-ver15

[2] [n.d.]. Cursors - Curse or blessing? https://social.msdn.microsoft.com/Forums/sqlserver/en-US/c34ea336-42cd-47ab-8dbe-6a1b7c0d5783/cursors-curse-or-blessing

[3] [n.d.]. Java Documentation: Retrieving and Modifying Values from Result Sets. https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html

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

[5] [n.d.]. Performance Considerations of Cursors. https://www.brentozar.com/sql-syntax-examples/cursor-example/

[6] [n.d.]. RUBBoS: Rice University Bulletin Board System Benchmark. http://jmob.ow2.org/rubbos.html

[7] [n.d.]. RUBiS: Rice University Bidding System Benchmark. http://rubis.ow2.org/

[8] [n.d.]. Scalar UDF Inlining. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining?view=sql-server-ver15

[9] [n.d.]. The Curse of the Cursors: Why You Don't Need Cursors in Code Development. https://www.datavail.com/blog/curse-of-the-cursors-in-code-development/

[10] [n.d.]. The Truth About Cursors. http://bradsruminations.blogspot.com/2010/05/truth-about-cursors-part-1.html

[11] [n.d.]. Transact SQL. https://docs.microsoft.com/en-us/sql/t-sql/language-elements/language-elements-transact-sql

[12] 2020. Adempiere: Opensource ERP Software. http://adempiere.net/

[13] 2020. Aggify Evaluation Workloads. http://aka.ms/WL-Aggify

[14] 2020. Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/

[15] 2020. LISTAGG: Oracle Database SQL Language Reference. https://docs.oracle.com/cd/E11882_01/server.112/e41084/functions089.htm#SQLRF30030

[16] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

[17] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[18] Alvin Cheung, Samuel Madden, Owen Arden, , and Andrew C Myers. 2012. Automatic Partitioning of Database Applications. In *Intl. Conf. on Very Large Databases*.

[19] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis *(PLDI)*. 3–14. https://doi.org/10.1145/2462156.2462180

[20] CLRU [n.d.]. CLR User-Defined Functions, https://msdn.microsoft.com/en-us/library/ms131077.aspx. https://msdn.microsoft.com/en-us/library/ms131077.aspx

[21] Sara Cohen. 2006. User-defined Aggregate Functions: Bridging Theory and Practice. In *ACM SIGMOD*. 49–60.

[22] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away. *arXiv e-prints*, Article arXiv:1909.03291 (Sep 2019), arXiv:1909.03291 pages. arXiv:cs.DB/1909.03291

[23] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications *(ACM SIGMOD)*. 16. https://doi.org/10.1145/2882903.2882926

[24] K Venkatesh Emani and S Sudarshan. 2018. Cobra: A Framework for Cost-Based Rewriting of Database Applications. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 689–700.

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[26] Sofoklis Floratos, Yanfeng Zhang, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2018. SQLoop: High Performance Iterative Processing in Data Management. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. 1039–1051. https://doi.org/10.1109/ICDCS.2018.00104

[27] César A. Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD*. 571–581. https://doi.org/10.1145/375663.375748

[28] S. Gupta, S. Purandare, and K. Ramachandra. 2020. Technical Report: Optimizing Cursor Loops In Relational Databases. *ArXiv e-prints* (April 2020). http://aka.ms/TR-Aggify

[29] Ravindra Guravannavar and S Sudarshan. 2008. Rewriting Procedures for Batched Bindings. In *Intl. Conf. on Very Large Databases*.

[30] HPLSQL [n.d.]. Procedural SQL on Hadoop, NoSQL and RDBMS. http://www.hplsql.org/why

[31] JDBC 2020. The Java Database Connectivity API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[32] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.

[33] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. 2009. *Data Flow Analysis: Theory and Practice*. CRC Press.

[34] Daniel Lieuwen and David DeWitt. 1992. A Transformation-Based Approach to Optimizing Loops in Database Programming Languages. *Sigmod Record* 21, 91–100. https://doi.org/10.1145/141484.130301

[35] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[36] Kisung Park, Hojin Seo, Mostofa Kamal Rasel, Young-Koo Lee, Chanho Jeong, Sung Yeol Lee, Chungmin Lee, and Dong-Hun Lee. 2019. Iterative Query Processing Based on Unified Optimization Techniques. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 54–68. https://doi.org/10.1145/3299869.3324960

[37] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating Imperative Code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 909–927. https://doi.org/10.1145/2660193.2660228

[38] 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. *PVLDB* 11, 4 (2017), 432–444.

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

[40] Weipeng P. Yan and Per bike Larson. 1995. Eager aggregation and lazy aggregation. In *In VLDB*. 345–357.