Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL

Kai Franz[♨], Samuel Arch[♨], Denis Hirn[♉], Torsten Grust[♉], Todd C. Mowry[♨], Andrew Pavlo[♨]
[♨]Carnegie Mellon University, [♉]University of Tübingen
udfs-cidr24@cs.cmu.edu

Abstract

SQL's user-defined functions (UDFs) allow developers to express complex computation using procedural logic. But UDFs have been the bane of database management systems (DBMSs) for decades because they inhibit optimization opportunities, potentially slowing down queries significantly. In response, *batching* and *inlining* techniques have been proposed to enable effective query optimization of UDF calls within SQL. Inlining is now available in a major commercial DBMS. But the trade-offs between both approaches on modern DBMSs remain unclear.

We evaluate and compare UDF batching and inlining on enterprise and open-source DBMSs using a state-of-the-art UDF-centric workload. We observe the surprising result that although inlining is better on simple UDFs, batching outperforms inlining by up to 93.4× for more complex UDFs because it makes it easier for a DBMS's query optimizer to decorrelate subqueries. We propose a hybrid approach that chooses batching or inlining to achieve the best performance.

1 SQL AND UDFs: A DIFFICULT AFFAIR

Nearly 50 years since the invention of SQL, it remains a challenge to express complex logic in a language that lacks procedural constructs such as sequential and branching control flow.

To address this problem, all the major DBMSs extend SQL with support for user-defined functions (UDFs) [19] written in procedural programming languages. UDFs were first introduced by INGRES in the 1970s to allow users to add operations on user-defined types [17]. Since SQL:1999, the SQL standard has included PL/PSM for defining UDFs. DBMSs such as Oracle, IBM DB2, PostgreSQL, and SQL Server support UDFs, particularly the kind that allow users to embed SQL queries inside UDFs (collectively referred to as PL/SQL UDFs in what follows). Such UDFs are now commonplace and are executed billions of times per day on Azure SQL Server alone [8, 19]. By encapsulating query logic inside UDFs and making nested UDF calls, UDFs allow for modularity, code reuse, and simplicity compared to pure SQL.

Despite their software engineering advantages, PL/SQL UDFs are notoriously difficult to optimize. The main challenge comes from the SQL queries that are intermixed with the UDF's procedural statements. Combining these two different programming paradigms makes it impossible for the query optimizer to reason about the UDF. As a result, UDFs become optimization barriers and the DBMS is forced to execute them iteratively, row-by-row, causing query execution times to explode. However, if the UDF were represented in a form that the query optimizer could understand, efficient set-oriented execution plans could be used instead:

Batching [9] is an optimization technique for UDFs that retains their procedural statement-by-statement execution mode. However, the DBMS executes each statement on the entire input simultaneously, taking advantage of efficient set-oriented query plans.

Inlining is an alternative optimization technique popularized by Froid [19] that eliminates the procedural statements by first translating the UDF into a logically equivalent relational representation (e.g., SQL) and then using it in the calling query as a correlated subquery. By expressing the entire computation as a single SQL query, this helps the optimizer find an efficient query plan.

Until now, an experimental comparison between different DBMSs was not possible because batching historically required a PL/SQL evaluator and Froid operates on SQL Server's internal relational algebra format. Although both techniques address the performance of UDFs, little is known about the relative strengths of these techniques and when one should be preferred over the other. To bridge this gap, we formally define source-to-source translations from PL/SQL to SQL for both batching and inlining, targeting any DBMS that supports SQL:1999's **LATERAL** joins.

Both batching and inlining produce subqueries for the DBMS to execute. However, naively executing these subqueries with correlated execution (for each row of the outer query, execute the subquery) results in disastrous performance. Therefore, DBMSs implement *subquery decorrelation* strategies that try to replace the subquery with a join operator instead, resulting in efficient execution plans. Our results show that batching often produces simpler subqueries that a DBMS can decorrelate, whereas it cannot do the same for inlined subqueries, ultimately resulting in batching outperforming inlining.

In this paper, we make the following contributions:

- We describe PL/SQL UDF to SQL translations for batching and inlining, enabling the first comparison of the two techniques.
- We evaluate UDF inlining and batching on multiple DBMSs and show that batching often outperforms inlining.
- We propose a hybrid strategy that determines whether to apply batching or inlining depending on the structure of the UDF.

 14th Annual Conference on Innovative Data Systems Research (CIDR '24). TODO, Chaminade, USA

```
CREATE FUNCTION getManufact(item_id INT) RETURNS CHAR(50) AS $$
BEGIN
  DECLARE man CHAR(50);
  DECLARE cnt1 INT; DECLARE cnt2 INT;
  man := '';
  cnt1 := (SELECT COUNT(*)                                        Q1[·]
           FROM   store_sales_history, date_dim
           WHERE  ss_item_sk = item_id
           AND    d_date_sk = ss_sold_date_sk AND d_year = 2023);

  cnt2 := (SELECT COUNT(*)                                        Q2[·]
           FROM   catalog_sales_history, date_dim
           WHERE  cs_item_sk = item_id
           AND    d_date_sk = cs_sold_date_sk AND d_year = 2023);

  IF (cnt1 > 0 AND cnt2 > 0)                                      Q3[·]
    THEN man := (SELECT i_manufact FROM item WHERE i_item_sk = item_id);
    ELSE man := 'outdated item';
  END IF;
  RETURN man;
END $$ LANGUAGE PLPGSQL;

-- Query calling UDF getManufact()
SELECT ws_item_sk                                  Q4[]       -- items sold by CompanyX
FROM  (SELECT ws_item_sk, COUNT(*) cnt                        -- on the web that are
       FROM web_sales                                         -- among the top 25k
       GROUP BY ws_item_sk
       ORDER BY cnt DESC, ws_item_sk                          -- UDF getManufact(item_id)
       LIMIT 25000) t1                                        -- identifies the item's
WHERE getManufact(ws_item_sk) = 'CompanyX';                   -- manufacturer in year 2023
```

Figure 1: UDF Example – UDF 20b from the ProcBench workload [8].

2 BACKGROUND ON PL/SQL UDFs

To understand the performance penalties of UDFs, we consider UDF 20b shown in Figure 1 from Microsoft's ProcBench [8] written in PostgreSQL's dialect of PL/SQL. For our analysis below, we assume a generic DBMS query engine implementation; we discuss system-specific optimizations in Section 4.

The query that invokes this UDF in Figure 1 retrieves items manufactured by 'CompanyX' that are among the top 25k highest-selling. At run time, the DBMS switches the execution context from the (declarative) query plan to the (procedural) PL/SQL interpreter. The system invokes the UDF by passing each row from the calling query as input. The UDF interpreter then evaluates each statement in the function sequentially, updating the state of local variables and returning the computed result. The DBMS then switches its execution context back to the query plan executor.

Suppose that a query invokes the UDF with `item_id = 1`. When the interpreter reaches Q1, it replaces `item_id` with the constant 1 and then executes the **SELECT** query. The DBMS will compute the store sales from 2023 through a semi-join (assuming no indexes) and applies the equality predicate on a large number of rows. This process repeats when the interpreter executes Q2. In effect, the DBMS executes a slow nested-loop join between the `web_sales` table from the calling query Q4 and the tables accessed in the UDF's **SELECT** queries. If a developer rewrote this query without the UDF and instead used only SQL, the DBMS could perform a more efficient hash or sort-merge join between these tables.

This problem occurs because of the impedance mismatch between declarative SQL and procedural UDFs. Slow UDF performance due to "row by agonizing row" (RBAR) execution has plagued developers for decades [10]. As a result, researchers have developed two methods for overcoming these issues: *batching* and *inlining*.

UDF Batching: Instead of invoking the UDF once per input row, the DBMS invokes the UDF once for multiple (possibly all) inputs at the same time. The idea is to transform each statement in the UDF into an **UPDATE** query that operates on all input values at once. The original batching proposal from 2008 requires modifying the DBMS to implement a PL/SQL evaluator to execute batched UDFs [9]. We present an alternative approach using pure SQL translation.

```
CREATE TEMPORARY TABLE state
( item INT, man CHAR(50), cnt1 INT, cnt2 INT, p BOOLEAN,
  res CHAR(50), returned BOOLEAN DEFAULT false, mult INT );

INSERT INTO state(item, mult)
  SELECT ws_item_sk, COUNT(*) AS mult
  FROM   Q4[]
  GROUP BY ws_item_sk;

UPDATE state SET man = '' WHERE NOT returned;
UPDATE state SET cnt1 = Q1[item] WHERE NOT returned;
UPDATE state SET cnt2 = Q2[item] WHERE NOT returned;
UPDATE state SET p = COALESCE(cnt1 > 0 AND cnt2 > 0, FALSE)
  WHERE NOT returned;
UPDATE state SET man = Q3[state.item] WHERE NOT returned AND p;
UPDATE state SET result = man, returned = true WHERE NOT returned AND p;
UPDATE state SET man = 'outdated item' WHERE NOT returned AND NOT p;
UPDATE state SET res = man, returned = true WHERE NOT returned AND NOT p;

SELECT s.item
FROM   state AS s, LATERAL generate_series(1, s.mult)
WHERE  s.res = 'CompanyX';
```

Figure 2: Batched UDF – A batched version of ProcBench UDF 20b [8].

```
SELECT ws_item_sk
FROM   Q4[]
WHERE (SELECT t4.retval
       FROM (SELECT '' AS man) AS t1(man),
       LATERAL (Q1[ws_item_sk]) AS t2(cnt1),
       LATERAL (Q2[ws_item_sk]) AS t3(cnt2),
       LATERAL (SELECT CASE WHEN t2.cnt1 > 0 AND t3.cnt2 > 0
                            THEN (Q3[ws_item_sk])
                            ELSE (SELECT 'outdated item')
                       END) AS t4(retval)) = 'CompanyX';
```

Figure 3: Inlined UDF – An inlined version of ProcBench UDF 20b [8].

Figure 2 shows the example UDF after transforming it into a batched form. To execute this batched UDF, the DBMS first creates a temporary **state** table with columns for the original UDF's scalar variables (e.g., **man**, **cnt1**). Each **state** row corresponds to the scalar variables for each input row. Additionally, there are dedicated columns in the **state** table that track the UDF's runtime status for the input row: (1) **returned** indicates whether the UDF execution has reached a return statement yet, (2) **res** stores the return value of the UDF, and (3) **mult** tracks the number of duplicates for each input.

Next, the DBMS executes a series of **UPDATE** queries that modify the **state** table in the same manner that each statement in the original UDF would update local variables. These **UPDATE** statements closely correspond to the original UDF statements but are expressed in SQL: assignments via **SET** clauses and conditionals via **WHERE** clauses. **SELECT** queries in the original UDF (i.e., Figure 1 – Q1) are referenced in the **SET** clause of the corresponding **UPDATE** statement in Figure 2. After executing every **UPDATE**, the **state** table contains the return value for each input row. To return the result back to the user, the DBMS executes a **SELECT** that references **state** with a **LATERAL** join on **mult** to recover the result for duplicate inputs.

By executing each statement of the UDF on an input batch, the DBMS leverages efficient join operators but incurs additional overhead from input materialization and manipulating temporary tables.

UDF Inlining: With this approach, the DBMS translates a UDF's statements to a logical plan (e.g., relational algebra, SQL) and then replaces (i.e., inlines) the UDF invocation in the calling query with the translated variant. This approach was first proposed by Simhadri et al. in 2014 by translating UDFs into relational algebra and inlining

it as a correlated subquery into the calling query at the logical plan level [21]. Inlining was further refined with the Microsoft Froid project [19] and subsequently included in SQL Server 2019 [3]. Microsoft's Aggify extended Froid to support inlining cursor loops by rewriting them as equivalent custom aggregates [7]. Inlining was further extended by Hirn et al. to add support for UDFs with arbitrary control flow using pure SQL translations [12].

Figure 3 shows UDF 20b after our inlining transformation has translated individual subqueries which are then chained together via **LATERAL** joins (or the **APPLY** operator in SQL Server) to express statement sequencing. Since later statements can see the assignments of their predecessor, these subqueries are correlated. Procedural **IF/ELSE** statements translate into **CASE/WHEN** expressions in SQL. Lastly, the translated UDF is substituted (i.e., inlined) into the calling query as a correlated subquery.

By representing the UDF in SQL, it is no longer an optimization barrier since the query optimizer can now reason about it. The DBMS can optimize across the query and the UDF, pipeline execution, leverage intra-query parallelism, and rely on efficient set-oriented execution plans. However, UDF performance is limited with inlining if the DBMS cannot decorrelate the subquery.

Trade-offs and Challenges: Although both batching and inlining facilitate efficient query execution across UDF boundaries, the two techniques have interesting trade-offs. A key advantage of inlining is that the DBMS can holistically optimize the query and execute it entirely pipelined without batching's materialization overheads. However, inlining's potential benefits crucially depend upon the DBMS's ability to decorrelate subqueries, which becomes more challenging as the UDFs become more complex. A subtle but important advantage of batching compared with inlining is that the resulting subqueries are simpler: they have at most a single **SELECT** query. As we will see in Section 4, batching outperforms inlining where the DBMS optimizer lacks support for robust decorrelation.

Achieving the necessary level of subquery decorrelation is hard, and researchers have been working on it for decades. The first techniques proposed in 1982 used SQL transformation rules to eliminate subqueries [14]. The next major milestone was Magic Set transformations from the 1990s [20]. Microsoft SQL Server's approach from 2001 introduces the **APPLY** operator enabling systematic decorrelation of subqueries through algebraic rewrites [5]; however, it cannot handle subqueries that require duplication of common subexpressions. Oracle's subquery decorrelation uses heuristic pattern matching to eliminate them, but it also suffers the same problem as SQL Server [2]. Neumann et al. [15] developed the first technique to systematically eliminate *all* subqueries algebraically [15]. To our knowledge, the only systems that implement this approach are HyPer [13], Umbra [16], Spark [23] and DuckDB [18].

3 FROM PL/SQL UDFs TO PLAIN SQL

We now describe how to use syntax-directed translations to transform PL/SQL-style UDFs into SQL to enable batching and inlining. Both transformations are applicable to any DBMS with contemporary SQL support but do not require UDF support.

Target input UDFs must adhere to the grammar of Figure 4. These UDFs are loop-free but may include statement sequences and branching control flow. Both translations target the compilation

f ≔ **CREATE FUNCTION** $v(v\ \tau,\ldots,v\ \tau)$
  RETURNS τ **AS BEGIN** d; s **END**;
d ≔ **DECLARE** $v\ \tau$
  | d; d
s ≔ **SET** v = a;
  | **IF** a s [**ELSE** s] **ENDIF**;
  | **RETURN** a;
  | **BEGIN** s **END**
  | s s
a ≔ *scalar SQL expression*
v ≔ ⟨*UDF/variable identifier*⟩
τ ≔ ⟨*scalar SQL type*⟩

Figure 4: SQL UDF Grammar.

of the imperative PL/SQL features and are indifferent to the syntax of embedded SQL queries (non-terminal a in Figure 4) of the underlying DBMS host: we never alter queries a but embed them unchanged into the generated SQL.

Batching ($\rightarrowtail_b$, Figure 5): Before the batching translation $\rightarrowtail_b$ applies, we expect (1) all PL/SQL variables in the input UDF to be declared upfront and (2) all predicate expressions a in conditional statements (**IF** a …**ENDIF**) to be assigned to variables (in UDF **getManufact** of Section 1 this leads to the additional declaration of a **BOOLEAN** variable **p** and assignment **SET p = cnt1 > 0 AND cnt2 > 0** in Line 13 of Figure 2).

The generated SQL code evaluates a UDF call **f(**x_1**,**…**,**x_n**)** by placing the arguments x_i into the columns v_i (representing **f**'s parameters) of temporary table **state** (Rule Batch in Figure 5). The additional columns *vars* and **res** of **state** hold the values of all UDF-local variables and the final result of **f**, respectively (all initialized with **NULL**). The Boolean column **returned** indicates whether UDF control flow has already reached a final **RETURN** (initially **false**).

Since batching aims to evaluate $m \gg 1$ UDF calls at once, table **state** will hold m rows of the above form, one for each call. If the ith UDF call is **f(**x_{1i}**,**…**,**x_{ni}**)**, then Rule Batch will populate **state** initially as shown here on the left. Lastly, should the ith UDF call happen $r_i > 1$ times with identical arguments, we collapse (*i.e.*, group) the associated duplicate rows in **state** and set column **mult** to r_i to record the repetition but evaluate the ith call only once.

state

v_1	⋯	v_n	*vars*	**res**	**returned**	**mult**
x_{11}	⋯	x_{n1}			false	r_1
⋮	⋱	⋮		NULL	⋮	⋮
x_{1m}	⋯	x_{nm}			false	r_m

Batching then maps PL/SQL assignment statement **SET** v **=** a to a SQL **UPDATE** statement on table **state**, setting column v to the result of embedded query a (Rule Assign). Sequences of such statements translate into sequences of individual **UPDATE**s (Rule Seq), each exhibiting a correlation between **state** and a at most, a query pattern simple enough to be in reach for most query optimizers. If a PL/SQL statement s lies in a conditional control flow path guarded by variable **v**, we guard the **UPDATE** statement for s by **WHERE** p **AND v** in which p represents all earlier control flow decisions of the call (see the Boolean context $p \vdash \cdots$ in the rules—initially, $p \equiv$ **NOT returned**, indicating that the UDF still executes). Statement **RETURN** a sets column **returned** to **true**, effectively disabling all further computation for that UDF call. After such a **RETURN**, column **ret** holds the final result a of the call (Rule Return).

Inlining ($\rightarrowtail_i$, Figure 6): The inlining translation $\rightarrowtail_i$ assumes that *all* control flow paths of the input UDF end in **RETURN** a and deliver a result value. Branching control flow paths thus never merge, allowing for a compact and simple rule set that avoids the use of SSA [1]. Bringing UDFs into this form may lead to statement duplication but, importantly, preserves control flow path length (i.e., UDF calls will *not* perform additional work at runtime).

Unlike batching, inlining $\rightarrowtail_i$ can disregard the UDF's PL/SQL variable declarations d and solely focuses on PL/SQL statements s.

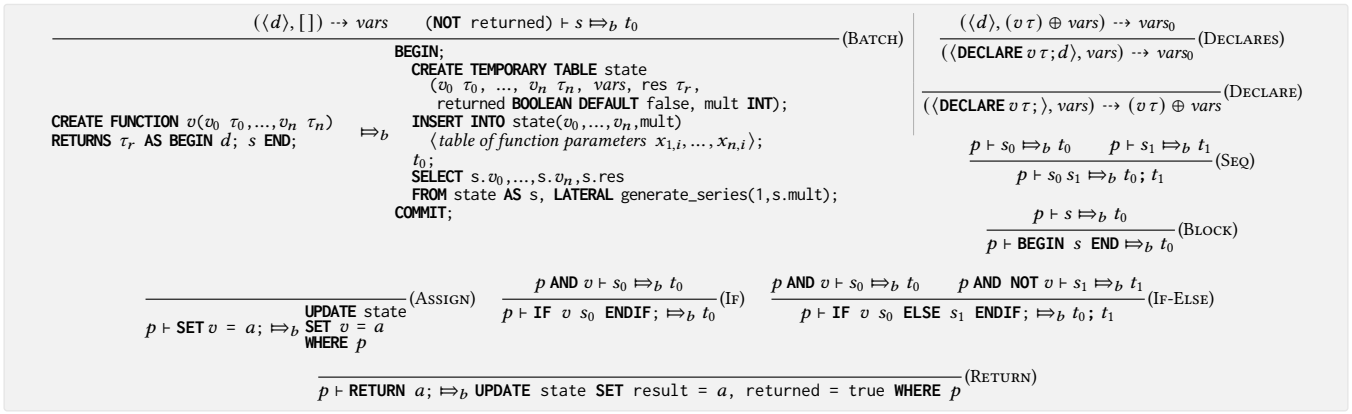


Figure 5: Batching Rule Set – Defines the UDF translation $\mapsto_b$ from PL/SQL to plain SQL (auxiliary mapping $\dashrightarrow$ collects declared PL/SQL variables).

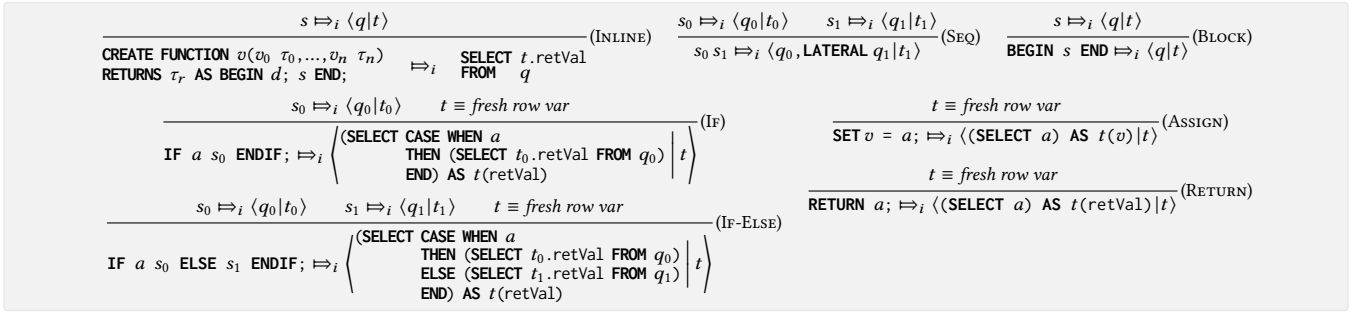


Figure 6: Inlining Rule Set – Defines the UDF translation $\mapsto_i$ from PL/SQL to plain SQL, following the Froid-style UDF inlining strategy.

We map statement s into the pair $\langle q \,\mathbf{AS}\, t(c) | t \rangle$ in which scalar SQL query q makes its result available in column $t.c$ (row variable t is made explicit to facilitate the translation of statement sequences, see below). Column $c \equiv v$ if s is the assignment $\mathbf{SET}\ v = a$ (Rule Assign in Figure 6), otherwise $c \equiv \mathbf{retVal}$, i.e., the statement's result value (Rules Return, If, If-Else). A statement sequence $s_0\, s_1$ is expressed in terms of SQL:1999's **LATERAL** (Rule Seq): SQL query q_1 for s_1 may access the values of variables bound by s_0's associated query q_0 through its row variable t_0. Sequences of n statements thus lead to a chain of $n-1$ **LATERAL** joins. In the absence of support for **LATERAL** joins, e.g. in backends like SQLite3, this required sequencing may alternatively be expressed through the correlated nesting of subqueries: the inner query q_1 may then depend on values computed by the outer query q_0. The decorrelation of either **LATERAL** joins or subqueries, however, may pose a challenge for some query engines if chains of length $n > 1$ are to be processed (see Section 4).

4 EVALUATION

For our comparison, we use ProcBench [8] with the default index configuration at a scale factor of 1 GB. ProcBench, which mirrors real-world PL/SQL usage, consists of query-UDF pairs built on the TPC-DS schema and data generator. Much like a TPC-DS query, a typical ProcBench query is read-only and makes use of scans, joins, and aggregates. As illustrated in Figure 1, a typical query starts with a simple query returning a large result set, followed by the invocation of a scalar UDF on each row. The invoked UDF scans several fact tables, filters based on its parameters, and returns a scalar aggregate of the filtered rows, occasionally with minor post-processing.

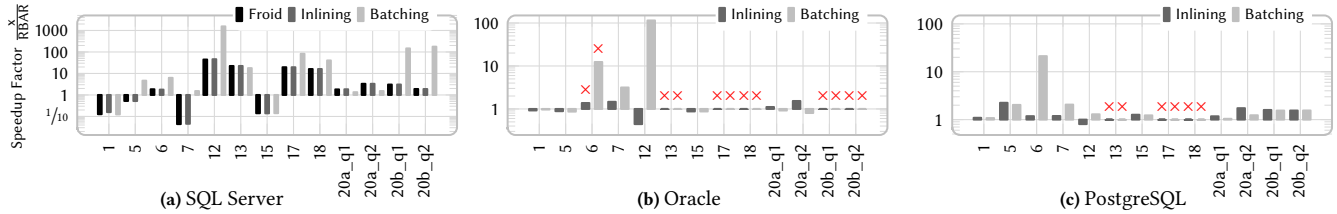
We only consider loop-free UDFs that (1) contain SQL statements, (2) have at least one input argument (otherwise, batching is not applicable), and (3) use the top-level UDF-invoking query defined by ProcBench. We also exclude UDFs 2 and 16 since they contain large cross products that do not terminate within 20 minutes on any DBMS. We also exclude UDF 19 because it is a table-valued function and therefore falls outside the scope of Froid-style inlining. Our workload contains 13/26 (50%) of the UDFs from ProcBench.

UDF 20 has two variants, a simple (20a) and a complex (20b) version: the complex version, depicted in Figure 1, checks that the item was sold in 2023 before looking up its manufacturer, while the simple version skips these checks and immediately executes Q_3.

Each variant of UDF 20 is invoked with two queries: variant q1 invokes the UDF on the items that were in the top 50k items sold in stores, online, and in catalogs, while q2 (depicted in Figure 1) invokes the UDF on the top 25k items sold online only.

For each DBMS[1], we tune their configuration knobs to achieve the best performance, pre-warm each DBMS's buffer pool, and refresh statistics. Each measurement is run five times, and the

[1]Hardware: Intel Xeon 5218R CPU with 192 GB DDR4 RAM, 500GB Samsung 970 EVO NVMe SSD. Hyper-threading disabled. All DBMSs were configured single-threaded to prevent parallelism, and pinned to a single NUMA region. DBMSs: (1) **SQL Server** v16.0.4045.3 (2) **Oracle 21c Enterprise** v21.3.0.0.0 (3) **DuckDB** Commit 40cb6d315b (4) **PostgreSQL** v15.3.



(a) SQL Server (b) Oracle (c) PostgreSQL

Figure 7: Batching vs. Inlining – Performance measurements for the ProcBench workload.

	Method	UDFs 1	5	6	7	12	13	15	17	18	20a_q1	20a_q2	20b_q1	20b_q2
SQL Server	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	Batched	✗	✓	(✓)	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
Oracle	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	Batched	✗	✗	(✓)	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
DuckDB	Inlined	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Batched	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PostgreSQL	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Batched	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 1: Subquery Decorrelation – Whether a given UDF's subqueries could be decorrelated by a DBMS after inlining or batching. Symbol (✓) indicates that some, but not all subqueries could be decorrelated.

average runtime is reported. When execution exceeds 20 minutes, we show an × atop the relevant bar.

4.1 Summary

Decorrelating the subqueries produced by the batched or inlined UDFs is critical to achieving high performance. Table 1 tracks whether each DBMS decorrelates the generated subqueries for each UDF. We observe that any UDF that is decorrelated with inlining is also decorrelated with batching (since the generated SQL is simpler).

The aggregate performance results in Table 2 show that batching improves performance more than inlining for the two enterprise DBMSs (SQL Server, Oracle). However, for simple UDFs, inlining is better. DuckDB decorrelates any subquery making inlining preferable (as materialization is avoided). PostgreSQL fails to decorrelate any subqueries, so neither technique is effective.

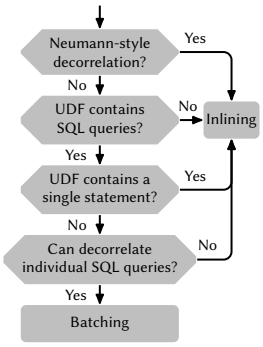


Figure 8: Hybrid Strategy for UDF Execution – Decides when to batch or inline a UDF.

Overall, a hybrid approach that uses batching when it performs best and inlining otherwise delivers the best performance. Our hybrid strategy is illustrated in the flowchart of Figure 8:

(1) If the DBMS uses Neumann-style decorrelation [15], use inlining to avoid materialization overheads.
(2) If the UDF does not contain embedded SQL queries, use inlining since decorrelation is not performance-critical.
(3) If the UDF contains a single statement, use inlining since batching produces a similar subquery.
(4) If the DBMS cannot decorrelate individual SQL queries then use inlining.
(5) Otherwise, use batching.

	Froid	Inlined	Batched	Hybrid
SQL Server	2.00×	2.04×	9.28×	10.46×
Oracle	N/A	1.00×	1.82×	1.94×
DuckDB	N/A	N/A	N/A	N/A
PostgreSQL	N/A	1.33×	1.85×	1.33×

Table 2: Performance Improvement – The geometric mean speedup for all UDFs over RBAR UDF execution. Froid is only available in SQL Server. There is no UDF support on DuckDB.

4.2 SQL Server

We first note that Figure 7(a) shows that inlining and Froid achieve nearly identical performance; this validates that our source-level translation to SQL is comparable to Microsoft's propietary Froid implementation. However, batching outperforms inlining for UDFs 5, 6, 7, 12, 17, 18, 20b_q1, and 20b_q2, with a speedup of up to 1550×. Our analysis of the query plans (Table 1) indicate that these are the UDFs whose embedded queries can be decorrelated with batching but not with inlining. We attribute this disparity to batching producing simpler subqueries for the DBMS to decorrelate compared to inlining: the latter produces a single `LATERAL`-chained subquery for the entire UDF. Inlining outperforms batching for UDFs 20a_q1 and 20a_q2 and achieves 3.3× better performance than RBAR. These UDFs contains a single `RETURN` statment such that their inlined transformation is sufficiently simple to be decorrelated by the DBMS: batching's materialization overhead is avoided. For the remaining UDFs (1, 7, 15), RBAR execution outperforms all three techniques due to a quirk in the optimizer that enables a bitmap semi-join optimization only for the RBAR query plan.

Summary: Batching outperforms inlining on SQL Server (9.28× speedup vs 2.04× speedup). However, the DBMS achieves the best performance (10.46× speedup) with a hybrid strategy.

4.3 Oracle

Next, Oracle's performance results in Figure 7(b) show that batching improves performance by up to 114×, outperforming inlining for UDFs 6, 7, and 12. Again, these UDFs are those that the DBMS decorrelates with batching but not with inlining (see Table 1). As explained in Section 2, Oracle is only able to decorrelate a subset of common subqueries. However, as with SQL Server, for simple UDFs (e.g., 20a_q1, 20a_q2), inlining is 1.53× faster than batching because it avoids materialization. For the remaining UDFs, batching and inlining both have a negligible effect compared to RBAR since the DBMS cannot decorrelate with either strategy.

Summary: Batching outperforms inlining on Oracle (1.82× vs 1.00×). Again, a hybrid strategy that uses batching if the DBMS decorrelates subqueries and inlining otherwise, results in the best

performance (1.94×). However, the performance gap between batching and inlining is less pronounced in Oracle than with SQL Server since even with batching, Oracle fails to decorrelate most UDFs.

4.4 DuckDB

Recall from Section 2 that DuckDB implements a state-of-the-art subquery decorrelation strategy that decorrelates *any* subquery [15]. Figure 9 shows that both batching and inlining result in efficient set-oriented query plans with no UDFs timing out. However, inlining outperforms batching, with up to 2209× speedups and a geometric mean speedup of 191×. We attribute this difference to batching's materialization overheads, whereas inlining completely pipelines query execution.

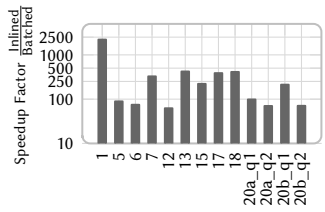


Figure 9: DuckDB – Inlining *vs.* batching (no RBAR on DuckDB).

Summary: For systems such as DuckDB that decorrelate arbitrary subqueries, inlining is superior to batching. Hence, one should always prefer inlining.

4.5 PostgreSQL

Finally, because PostgreSQL cannot decorrelate any of the UDF's subqueries, the performance difference between RBAR, batching, and inlining is minimal for almost all UDFs. However, we observe that for some UDFs, batching still outperforms inlining because it avoids UDF recomputation for duplicate inputs (as described in Section 3). This optimization could also be applied to inlining and is not specific to batching. For UDF 6, 94% of the inputs are duplicates, making batching 17.94× faster than inlining. Similarly, for UDF 7, 33% of the inputs are duplicates, making it 1.73× faster than inlining. Finally, UDF 12 is 1.61× faster with batching than inlining because **CASE WHEN** expressions cause redundant recalculation. Despite these performance optimizations, batching and inlining are ineffective since PostgreSQL's subquery decorrelation is too weak.

Summary: PostgreSQL represents our worst-case scenario for UDF optimizations. Its query optimizer is the most ineffective at subquery decorrelation of all the DBMSs that we have evaluated. Thus, for DBMSs that cannot decorrelate subqueries, neither inlining nor batching will be effective. Better support for subquery decorrelation is needed to address UDF performance.

5 RELATED WORK

An alternative technique to improve UDF performance is *compilation*, whereby the UDF is translated to a lower-level language with higher efficiency. Tuplex uses speculative compilation techniques to evaluate Python UDFs [22]. YeSQL improves upon Tuplex with tracing JIT compilation and UDF Fusion [4]. BabelFish employs holistic optimizations to efficiently execute "polyglot queries" (i.e., queries invoking UDFs written in multiple languages) [6]. But even with compilation UDFs remain as barriers to the query optimizer.

Froid [19] demonstrated significant performance benefits of UDF inlining in SQL Server, with speedup factors from 5 to 1000 on customer workloads. For ProcBench, however, we observed a geometric mean speedup of 2.04×, compared to 9.28× for batching, and 10.46× for our hybrid approach. This discrepancy can be attributed to two key factors: (1) parallelism and (2) UDF complexity.

Parallelism: When SQL Server evaluates a SQL query that invokes a UDF, parallelism for that query is disabled to guarantee sound UDF execution. Other DBMSs, like PostgreSQL and Oracle, provide **PARALLEL SAFE** annotations to mark UDFs that do not interfere with parallel evaluation. Thus, the only source of parallelism are the individual SQL queries embedded in the UDF provided that the UDF does not contain procedural statements only. Inlining as performed by Froid or our transformation $\Rightarrow_i$ (Figure 6), unlocks full intra-query parallelism, resulting in almost linear speedups relative to the number of threads. To focus on inlining's impact, we controlled for parallelism by running all DBMSs single-threaded. The Froid experiments use a CPU with 12 threads, which may account for the 12× speedup in their results [19].

UDF Complexity: On Azure SQL Server, the average T-SQL UDF contains 0.65 SQL queries [8]. Inlining can provide excellent performance on SQL Server for many of these real-world UDFs. ProcBench contains more complex UDFs (on average 2.15 SQL queries per UDF). The DBMS struggles to decorrelate the inlined UDFs, making inlining less effective in our experiments than in the original Froid paper. Using batching in these cases and inlining otherwise results in the best overall performance (10.46× speedup) compared to inlining alone (2.04× speedup).

6 CONCLUSIONS

Inlining and batching can be incredibly effective at improving UDF performance, but only if the DBMS can decorrelate the subqueries they produce. Because batching produces simpler subqueries than inlining, batching can surprisingly outperform inlining for many of the UDFs in our experiments. The best performance is achieved with a hybrid strategy, which is detailed in Figure 8.

There are several exciting research challenges to explore in the space of UDFs. Firstly, developing new translation techniques beyond batching and inlining tailored to the DBMS subquery decorrelation can achieve even better performance. Another exciting direction involves new UDF-to-SQL translations that generate **LATERAL**-free code, unlocking UDF support for DBMSs such as SQLite [11] that lack support for **LATERAL** joins. Another unexplored direction involves tactfully combining UDF inlining and compilation to get the best of both worlds. Lastly, modern programming languages such as Python, JavaScript, or WebAssembly are growing in popularity among users, and further research is required to integrate them into the query optimization pipeline of DBMSs.

References

[1] B. Alpern et al. 1988. Detecting Equality of Values in Programs. In *POPL*.
[2] S. Bellamkonda et al. 2009. Enhanced subquery optimizations in oracle. *VLDB* (2009).
[3] E. Darling. 2022. When Does Scalar UDF Inlining Work In SQL Server? https://erikdarlingdata.com/when-does-udf-inlining-kick-in/.
[4] Y. Foufoulas et al. 2022. YeSQL: " you extend SQL" with rich and highly performant user-defined functions in relational databases. *VLDB* (2022).
[5] C. Galindo-Legaria et al. 2001. Orthogonal optimization of subqueries and aggregation. *SIGMOD Record* (2001).

[6] P. Grulich et al. 2021. Babelfish: Efficient execution of polyglot queries. *VLDB* (2021).
[7] S. Gupta et al. 2020. Aggify: Lifting the curse of cursor loops using custom aggregates. In *SIGMOD*.
[8] S. Gupta et al. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *VLDB* (2021).
[9] R. Guravannavar et al. 2008. Rewriting procedures for batched bindings. *VLDB* (2008).
[10] C. Heinzelman. 2011. Unintended Consequences of Scalar-Valued User Defined Functions. https://learn.microsoft.com/en-us/archive/blogs/sqlcat/unintended-consequences-of-scalar-valued-user-defined-functions.
[11] Richard D Hipp. 2020. SQLite. https://www.sqlite.org/index.html
[12] D. Hirn et al. 2021. One with recursive is worth many GOTOs. In *SIGMOD*.
[13] A. Kemper et al. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*.
[14] W. Kim. 1982. On optimizing an SQL-like nested query. *TODS* (1982).
[15] T. Neumann et al. 2015. Unnesting arbitrary queries. *BTW* (2015).
[16] T. Neumann et al. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.
[17] J. Ong et al. 1983. Implementation of data abstraction in the relational database system ingres. *SIGMOD Record* (1983).
[18] M. Raasveldt et al. 2019. Duckdb: an embeddable analytical database. In *SIGMOD*.
[19] K. Ramachandra et al. 2017. Froid: Optimization of imperative programs in a relational database. *VLDB* (2017).
[20] P. Seshadri et al. 1996. Complex query decorrelation. In *ICDE*.
[21] V. Simhadri et al. 2014. Decorrelation of user defined function invocations in queries. In *ICDE*.
[22] L. Spiegelberg et al. 2021. Tuplex: Data science in Python at native code speed. In *SIGMOD*.
[23] A. Wang. 2023. https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/DecorrelateInnerQuery.