LAST CLASS

We covered two category of join algorithms for modern OLAP DBMSs.

**Hash Joins:** Every DBMS does this now

**Worst-case Optimal Joins:** Every DBMS will need to do something like this in the future.
EMBEDDED DATABASE LOGIC

Moving application logic into the DBMS can (potentially) provide several benefits:
→ Fewer network round-trips (better efficiency).
→ Immediate notification of changes.
→ DBMS spends less time waiting during transactions.
→ Developers do not have to reimplement functionality.
→ Extend the functionality of the DBMS.
EMBEDDED DATABASE LOGIC

User-Defined Functions (UDFs)
Stored Procedures
Triggers
User-Defined Types (UDTs)
User-Defined Aggregates (UDAs)

PROCEDURAL EXTENSIONS OF SQL:
UNDERSTANDING THEIR USAGE IN THE WILD
VLDB 2021
A **user-defined function** (UDF) is a function written by the application developer that extends the system's functionality beyond its built-in operations.

→ It takes in input arguments (scalars)
→ Perform some computation
→ Return a result (scalars, tables)
USER-DEFINED FUNCTIONS

Application

```python
def func1(args):
    execute(SQL)
    <Program Logic>
    return (result)
```

```python
def func2(args):
    <Program Logic>
    return (result)
```
USER-DEFINED FUNCTIONS

Application

execute(SQL)
execute(SQL)

```
SELECT * FROM xxx
WHERE val = func1(id)
```
TODAY’S AGENDA

Background
UDF In-lining
UDF CTE Conversion
UDF Batching
A SQL-based UDF contains a list of queries that the DBMS executes in order when invoked.
→ The function returns the result of the last query executed.

```
CREATE FUNCTION get_foo(int) RETURNS foo 
LANGUAGE SQL AS $$
SELECT * FROM foo WHERE foo.id = $1;
$$;
```

*Input Args*
UDF: SQL FUNCTIONS

A SQL-based UDF contains a list of queries that the DBMS executes in order when invoked.
→ The function returns the result of the last query executed.

```sql
CREATE FUNCTION get_foo(int)
RETURNS foo
LANGUAGE SQL AS $$
SELECT * FROM foo WHERE foo.id = $1;$$;
```

Return Args
UDF: SQL FUNCTIONS

A SQL-based UDF contains a list of queries that the DBMS executes in order when invoked.
→ The function returns the result of the last query executed.

```sql
CREATE FUNCTION get_foo(int)
    RETURNS foo
    LANGUAGE SQL AS $$
    SELECT * FROM foo WHERE foo.id = $1;
$$;
```

Function Body
UDF: SQL FUNCTIONS

A SQL-based UDF contains a list of queries that the DBMS executes in order when invoked.
→ The function returns the result of the last query executed.

CREATE FUNCTION get_foo(int)
    RETURNS foo
    LANGUAGE SQL AS $$
    SELECT * FROM foo WHERE foo.id = $1;
    $$;

SELECT get_foo(1);
SELECT * FROM get_foo(1);
UDF: EXTERNAL PROGRAMMING LANGUAGE

Some DBMSs support writing UDFs in languages other than SQL.
→ SQL Standard: SQL/PSM
→ Oracle/DB2: PL/SQL
→ Postgres: PL/pgSQL
→ DB2: SQL PL
→ MSSQL/Sybase: Transact-SQL

Other systems support more common programming languages:
→ Sandbox vs. non-Sandbox
CREATE FUNCTION cust_level(@ckey int) RETURNS char(10) AS 
BEGIN
    DECLARE @total float;
    DECLARE @level char(10);
    SELECT @total = SUM(o_totalprice) 
        FROM orders WHERE o_custkey=@ckey;
    IF (@total > 1000000)
        SET @level = 'Platinum';
    ELSE
        SET @level = 'Regular';
    RETURN @level;
END

SELECT c_custkey, cust_level(c_custkey) 
FROM customer

Get all the customer ids and compute their customer service level based on the amount of money they have spent.
**UDF ADVANTAGES**

They encourage modularity and code reuse
→ Different queries can reuse the same application logic without having to reimplement it each time.

Fewer network round-trips between application server and DBMS for complex operations.

Some types of application logic are easier to express and read as UDFs than SQL.
Query optimizers treat external programming language UDFs as black boxes.

→ DBMS is unable to estimate the function's cost / selectivity if it doesn't understand what the logic inside of it will do when it runs.
→ Example: `WHERE val = my_udf(123)`

It is difficult to parallelize UDFs due to correlated queries inside of them.

→ Some DBMSs will only execute queries with a single thread if they contain a UDF.
→ Some UDFs incrementally construct queries.
UDF DISADVANTAGES (2)

Complex UDFs in **SELECT / WHERE** clauses force the DBMS to execute iteratively.

→ RBAR = "Row By Agonizing Row"

→ Things get even worse if UDF invokes queries due to implicit joins that the optimizer cannot "see".

Since the DBMS executes the commands in the UDF one-by-one, it is unable to perform cross-statement optimizations.
UDF DISADVANTAGES (2)

Complex UDFs in SELECT/WHERE clauses force the DBMS to execute iteratively.

→ RBAR = “Row By Agonizing Row”

→ Things get even worse if UDF invokes queries due to implicit joins that the optimizer cannot “see”.

Since the DBMS executes the commands in the UDF one-by-one, it is unable to perform cross-

TSQL Scalar functions are evil.

I’ve been working with a number of clients recently who all have suffered at the hands of TSQL Scalar functions. Scalar functions were introduced in SQL 2000 as a means to wrap logic so we benefit from code reuse and simplify our queries. Who would be daft enough not to think this was a good idea. I for one jumped on this initially thinking it was a great thing to do.

However as you might have gathered from the title scalar functions aren’t the nice friend you may think they are.

If you are running queries across large tables then this may explain why you are getting poor performance.

In this post we will look at a simple padding function, we will be creating large volumes to emphasize the issue with scalar udfs.

```sql
create function PadLeft(@val varchar(100), @len int, @char char(1))
returns varchar(100)
as
begin
    return right(replac(@char, @len) + @val, @len)
end
go
```

Interpreted

Scalar functions are interpreted code that means EVERY call to the function results in your code being interpreted. That means overhead for processing your function is proportional to the number of rows.

Running this code you will see that the native system calls take considerably less time than the UDF calls. On my machine it takes 2814 ms for the system calls and 38758ms for the UDF. That’s a 14x increase.

```sql
set statistics time on
go
select max(right(repl大量('a', 100) + o.name + c.name, 100))
from msdb.sys.columns o
cross join msdb.sys.columns c
```

```sql
select max(db2.PadLeft(o.name + c.name, 100,'0'))
from msdb.sys.columns o
cross join msdb.sys.columns c
```
UDF DISADVANTAGES (2)

TSQL Scalar functions

I’ve been working with a number of clients recently who all have suffered with UDFs in SQL 2000 as a means to wrap logic so we benefit from code reuse and good idea. I for one jumped on this initially thinking it was a great thing.

However as you might have gathered from the title scalar functions

If you are running queries across large tables then this may explain:

In this post we will look at a simple padding function, we will be creating:

```
create function PadLeft(@val varchar(100), @len int) returns varchar(100)
as
begin
    return right(replicate(' ', @len) + @val, @len);
end
```

go

Interpreted

Scalar functions are interpreted code that means EVERY call to the function processing your function is proportional to the number of rows it processes.

Running this code you will see that the native system calls take 413 ms and the interpreted takes 3873 ms for the UDF. That’s a 9x increase.

```
set statistics time on
select max(right(replicate(' ',100) + o.name, 100))
from msdb.sys.columns o
cross join msdb.sys.columns c
```

```
set statistics time on
select max(dbp.PadLeft(o.name + c.name, 100))
from msdb.sys.columns o
cross join msdb.sys.columns c
```

Soften the RBAR impact with Native Compiled UDFs in SQL Server 2016

First published on MSDN on Feb 17, 2016

Keywords: Joa Sack, Danilo Ribiero, Toshio Kogoj

Many of us are very familiar with the negative performance implications of using scalar UDFs on columns in queries. My colleagues have posted about issues here and here. Using UDFs in this manner is an anti-pattern, most of us know upon because the row-by-row agonizing row (RBAR) processing that this implies. In addition, using UDFs also limits the optimizer to use serial plans. Overall, evil personified!

Native Compiled UDFs introduced

Though the problem with scalar UDFs is well known, we still come across workloads where this problem is a serious detriment to the performance of the query. In some cases, it may be easy to refactor the UDF as an inline Table Valued Function. But in other cases, it may simply not be possible to refactor the UDF.

SQL Server 2016 offers natively compiled UDFs, which can be of interest where refactoring the UDF to a TVF is not possible, or where the number of referring T-SQL objects are simply too many. Natively compiled UDFs will NOT eliminate the RBAR agony, but they can make each iteration incrementally faster.

Real-life results

We recently worked with an actual customer workload in the lab. In this workload, we had a query which invoked a scalar UDF in the output list. That means the UDF was actually executing once per row – in this case a total of 75 million rows! The UDF has a simple CASE expression inside it. However, we wanted to improve query performance so we decided to try rewriting the UDF.

We found the following results with the trivial UDF being refactored as a TVF versus the same UDF being natively compiled (all timings are in milliseconds):

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial UDF</td>
<td>750,000</td>
</tr>
<tr>
<td>Native Compiled UDF</td>
<td>600,000</td>
</tr>
<tr>
<td>Native UDF</td>
<td>250,000</td>
</tr>
</tbody>
</table>

15-721 (Spring 2024)
TPC-H Q12 using a UDF (SF=1).

→ **Original Query:** 0.8 sec
→ **Query + UDF:** 13 hr 30 min

```sql
SELECT l_shipmode, 
  SUM(CASE 
    WHEN o_orderpriority <> '1-URGENT' 
    THEN 1 ELSE 0 END) 
  AS low_line_count 
FROM orders, lineitem 
WHERE o_orderkey = l_orderkey 
  AND l_shipmode IN ('MAIL', 'SHIP') 
  AND l_commitdate < l_receiptdate 
  AND l_shipdate < l_commitdate 
  AND l_receiptdate >= '1994-01-01' 
  AND dbo.cust_name(o_custkey) IS NOT NULL 
GROUP BY l_shipmode 
ORDER BY l_shipmode
```

```sql
CREATE FUNCTION cust_name(@ckey int) 
RETURNS char(25) AS 
BEGIN 
  DECLARE @n char(25); 
  SELECT @n = c_name 
  FROM customer WHERE c_custkey = @ckey; 
  RETURN @n; 
END
```
UDF ACCELERATION

Approach #1: Compilation
→ Compile interpreted UDF code into native machine code.

Approach #2: Parallelization
→ Rely on user-defined annotations to determine which portions of a UDF can be safely executed in parallel.

Approach #3: Inlining
→ Convert UDF into declarative form and then inline it into the calling query.

Approach #4: Batching
→ Convert a UDF into corresponding SQL queries that operate on multiple tuples at a time.

Source: Surabhi Gupta
FROID UDF INLINING

Automatically convert UDFs into relational algebra expressions that are inlined as sub-queries. → Does not require the app developer to change UDF code.

Perform conversion during the rewrite phase to avoid having to change the cost-base optimizer. → Commercial DBMSs already have powerful transformation rules for executing sub-queries efficiently.
The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:
- Rewrite to de-correlate and/or flatten them
- Decompose nested query and store result to temporary table. Then the outer joins with the temporary table.

We will cover the German-style de-correlation for sub-queries next week...
Example: Retrieve the first user that has made at least two purchases.

```sql
SELECT user_id FROM orders AS o1
WHERE EXISTS(
    SELECT COUNT(*) FROM orders AS o2
    WHERE o1.user_id = o2.user_id
    GROUP BY o2.user_id HAVING COUNT(*) >= 2
)
ORDER BY user_id ASC LIMIT 1;
```

Rewritten query:

```sql
SELECT user_id FROM orders
GROUP BY user_id
HAVING COUNT(*) >= 2
ORDER BY user_id ASC LIMIT 1;
```
A lateral inner subquery can refer to fields in rows of the table reference to determine which rows to return.
→ Allows you to have sub-queries in FROM clause.

The DBMS iterates through each row in the table referenced and evaluates the inner sub-query for each row.
→ The rows returned by the inner sub-query are added to the result of the join with the outer query.
Example: Retrieve the first user that has made at least two purchases along with the timestamps of the first and next orders.

```
SELECT user_id, first_order, next_order, id
FROM (SELECT user_id,
          MIN(created) AS first_order
       FROM orders
       GROUP BY user_id) AS o1
INNER JOIN LATERAL
(SELECT id, created AS next_order
 FROM orders
 WHERE user_id = o1.user_id
 AND created > o1.first_order
 ORDER BY created ASC LIMIT 1) AS o2
ON true
LIMIT 1;
```
Example: Retrieve the first user that has made at least two purchases along with the timestamps of the first and next orders.

```sql
SELECT user_id, first_order, next_order, id
FROM (SELECT user_id, MIN(created) AS first_order
      FROM orders
      GROUP BY user_id) AS o1
INNER JOIN LATERAL
  (SELECT id, created AS next_order
   FROM orders
   WHERE user_id = o1.user_id
   AND created > o1.first_order
   ORDER BY created ASC LIMIT 1) AS o2
ON true
LIMIT 1;
```

Source: Krzysztof Kempiński
FROID OVERVIEW

Step #1 – Transform Statements
Step #2 – Break UDF into Regions
Step #3 – Merge Expressions
Step #4 – Inline UDF Expression into Query
Step #5 – Run Updated Query through Optimizer
**STEP #1: TRANSFORM STATEMENTS**

**Imperative Statements**

- `SET @level = 'Regular';`
- `SELECT @total = SUM(o_totalprice) FROM orders WHERE o_custkey=@ckey;`
- `IF (@total > 1000000)
  SET @level = 'Platinum';`

**SQL Statements**

- `SELECT 'Regular' AS level;`
- `SELECT (SELECT SUM(o_totalprice) FROM orders WHERE o_custkey=@ckey) AS total;`
- `SELECT (CASE WHEN total > 1000000 THEN 'Platinum' ELSE NULL END) AS level;`

Source: Karthik Ramachandra
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
    DECLARE @total float;
    DECLARE @level char(10);

    SELECT @total = SUM(o_totalprice) 
    FROM orders WHERE o_custkey=@ckey;

    IF (@total > 1000000)
        SET @level = 'Platinum';
    ELSE
        SET @level = 'Regular';

    RETURN @level;
END

(SELECT NULL AS level, 
(SELECT SUM(o_totalprice) 
FROM orders WHERE o_custkey=@ckey) AS total) AS E_R1
CREATE FUNCTION cust_level(@ckey int) RETURNS char(10) AS
BEGIN
    DECLARE @total float;
    DECLARE @level char(10);
    SELECT @total = SUM(o_totalprice)
       FROM orders
       WHERE o_custkey=@ckey;
    IF (@total > 1000000)
        SET @level = 'Platinum';
    ELSE
        SET @level = 'Regular';
    RETURN @level;
END

(SELECT NULL AS level,
 (SELECT SUM(o_totalprice)
       FROM orders
       WHERE o_custkey=@ckey) AS total)
) AS E_R1
CREATE FUNCTION `cust_level`(@ckey int) 
RETURNS char(10) AS 
BEGIN

DECLARE @total float;
DECLARE @level char(10);

SELECT @total = SUM(o_totalprice) 
FROM orders 
WHERE o_custkey = @ckey;

IF (@total > 1000000) 
SET @level = 'Platinum';
ELSE 
SET @level = 'Regular';

RETURN @level;
END

(SELECT NULL AS level,
(SELECT SUM(o_totalprice) 
FROM orders 
WHERE o_custkey = @ckey) AS total) AS E_R1

(SELECT ( 
CASE WHEN E_R1.total > 1000000 
THEN 'Platinum' 
ELSE E_R1.level END) AS level) AS E_R2
CREATE FUNCTION cust_level(@ckey int) RETURNS char(10) AS
BEGIN

DECLARE @total float;
DECLARE @level char(10);

SELECT @total = SUM(o_totalprice)
FROM orders
WHERE o_custkey = @ckey;

IF (@total > 1000000)
SET @level = 'Platinum';
ELSE
SET @level = 'Regular';

RETURN @level;
END

(SELECT NULL AS level,
(SELECT SUM(o_totalprice)
FROM orders
WHERE o_custkey = @ckey) AS total)
AS E_R1

(SELECT (CASE WHEN E_R1.total > 1000000
THEN 'Platinum'
ELSE E_R1.level END) AS level)
AS E_R2

(SELECT (CASE WHEN E_R1.total <= 1000000
THEN 'Regular'
ELSE E_R2.level END) AS level)
AS E_R3
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
DECLARE @total float;
DECLARE @level char(10);
SELECT @total = SUM(o_totalprice)
FROM orders WHERE o_custkey=@ckey;
IF (@total > 1000000)
SET @level = 'Platinum';
ELSE
SET @level = 'Regular';
RETURN @level;
END

(SELECT NULL AS level,
(SELECT SUM(o_totalprice)
FROM orders WHERE o_custkey=@ckey) AS total)
AS E_R1

(SELECT (CASE WHEN E_R1.total > 1000000
THEN 'Platinum'
ELSE E_R1.level END) AS level)
AS E_R2

(SELECT (CASE WHEN E_R1.total <= 1000000
THEN 'Regular'
ELSE E_R2.level END) AS level)
AS E_R3
**STEP #3: MERGE EXPRESSIONS**

```
(SELECT NULL AS level,
 (SELECT SUM(o_totalprice)
 FROM orders
 WHERE o_custkey=@ckey) AS total
 ) AS E_R1

(SELECT (  
 CASE WHEN E_R1.total > 1000000  
 THEN 'Platinum'
 ELSE E_R1.level END) AS level
 ) AS E_R2

(SELECT (  
 CASE WHEN E_R1.total <= 1000000  
 THEN 'Regular'
 ELSE E_R2.level END) AS level
 ) AS E_R3

SELECT E_R3.level FROM
 (SELECT NULL AS level,
 (SELECT SUM(o_totalprice)
 FROM orders
 WHERE o_custkey=@ckey) AS total
 ) AS E_R1
 CROSS APPLY
 (SELECT (  
 CASE WHEN E_R1.total > 1000000  
 THEN 'Platinum'
 ELSE E_R1.level END) AS level
 ) AS E_R2
 CROSS APPLY
 (SELECT (  
 CASE WHEN E_R1.total <= 1000000  
 THEN 'Regular'
 ELSE E_R2.level END) AS level
 ) AS E_R3;
```
STEP #3: MERGE EXPRESSIONS

```
(SELECT NULL AS level,
(SELECT SUM(o_totalprice)
  FROM orders
  WHERE o_custkey=@ckey) AS total
) AS E_R1

(SELECT (
    CASE WHEN E_R1.total > 1000000
      THEN 'Platinum'
      ELSE E_R1.level END) AS level
) AS E_R2

(SELECT (
    CASE WHEN E_R1.total <= 1000000
      THEN 'Regular'
      ELSE E_R2.level END) AS level
) AS E_R3

SELECT E_R3.level FROM
(SELECT NULL AS level,
(SELECT SUM(o_totalprice)
  FROM orders
  WHERE o_custkey=@ckey) AS total
) AS E_R1
CROSS APPLY
(SELECT (
    CASE WHEN E_R1.total > 1000000
      THEN 'Platinum'
      ELSE E_R1.level END) AS level
) AS E_R2
CROSS APPLY
(SELECT (
    CASE WHEN E_R1.total <= 1000000
      THEN 'Regular'
      ELSE E_R2.level END) AS level
) AS E_R3;
```
**STEP #4: INLINE EXPRESSION**

**Original Query**

```
SELECT c_custkey,
cust_level(c_custkey)
FROM customer
```

```
SELECT c_custkey,
    (SELECT E_R3.level FROM
        (SELECT NULL AS level,
            (SELECT SUM(o_totalprice)
                FROM orders
                WHERE o_custkey=@ckey)
            AS total
        ) AS E_R1
    CROSS APPLY
        (SELECT (CASE WHEN E_R1.total > 1000000
                     THEN 'Platinum'
                     ELSE E_R1.level END) AS level
        ) AS E_R2
    CROSS APPLY
        (SELECT (CASE WHEN E_R1.total <= 1000000
                     THEN 'Regular'
                     ELSE E_R2.level END) AS level
        ) AS E_R3;
) FROM customer;
```
**STEP #4: INLINE EXPRESSION**

**Original Query**

```
SELECT c_custkey,
      cust_level(c_custkey)
FROM customer
```

**Transformed Query**

```
SELECT c_custkey, (SELECT E_R3.level FROM (SELECT NULL AS level,
                                              (SELECT SUM(o_totalprice) FROM orders
                                              WHERE o_custkey=@ckey) AS total)
                                  AS E_R1
                          CROSS APPLY (SELECT (CASE WHEN E_R1.total > 1000000
                                                     THEN 'Platinum'
                                                     ELSE E_R1.level END) AS level
                                          ) AS E_R2
                      CROSS APPLY (SELECT (CASE WHEN E_R1.total <= 1000000
                                                 THEN 'Regular'
                                                 ELSE E_R2.level END) AS level
                          ) AS E_R3;
) FROM customer;
```
STEP #5: OPTIMIZE

SELECT c.c_custkey, (SELECT E_R3.level FROM (SELECT NULL AS level, (SELECT SUM(o_totalprice) FROM orders WHERE o_custkey = @ckey) AS total) AS E_R1 CROSS APPLY (SELECT (CASE WHEN E_R1.total > 1000000 THEN 'Platinum' ELSE E_R1.level END) AS level) AS E_R2 CROSS APPLY (SELECT (CASE WHEN E_R1.total <= 1000000 THEN 'Regular' ELSE E_R2.level END) AS level) AS E_R3) FROM customer;

SELECT c.c_custkey, (CASE WHEN e.total > 1000000 THEN 'Platinum' ELSE 'Regular' END) FROM customer c LEFT OUTER JOIN (SELECT o_custkey, SUM(o_totalprice) AS total FROM order GROUP BY o_custkey) AS e ON c.c_custkey = e.o_custkey;
CREATE FUNCTION `getVal`(@x int)
RETURNS char(10) AS
BEGIN
DECLARE @val char(10);
IF (@x > 1000)
SET @val = 'high';
ELSE
SET @val = 'low';
RETURN @val + ' value';
END

SELECT `getVal`(5000);
CREATE FUNCTION getVal(@x int)
RETURNS char(10) AS
BEGIN
DECLARE @val char(10);
IF (@x > 1000)
SET @val = 'high';
ELSE
SET @val = 'low';
RETURN @val + ' value';
END

Dynamic Slicing

BEGIN
DECLARE @val char(10);
SET @val = 'high';
RETURN @val + ' value';
END

Constant Propagation & Folding

BEGIN
DECLARE @val char(10);
SET @val = 'high value';
RETURN @val;
END

Dead Code Elimination

BEGIN
RETURN 'high value';
END

SELECT returnVal FROM
(SELECT CASE WHEN @x > 1000
THEN 'high'
ELSE 'low' END AS val)
AS E_R1
OUTER APPLY
(SELECT DT1.val + ' value'
AS E_R2)
AS E_R2

SELECT returnVal FROM
(SELECT 'high' AS val)
AS E_R1
OUTER APPLY
(SELECT 'high value' AS returnVal)
AS E_R2

SELECT 'high value';
SUPPORTED OPERATIONS (2019)

T-SQL Syntax:
→ DECLARE, SET (variable declaration, assignment)
→ SELECT (SQL query, assignment)
→ IF / ELSE / ELSE IF (arbitrary nesting)
→ RETURN (multiple occurrences)
→ EXISTS, NOT EXISTS, ISNULL, IN, … (Other relational algebra operations)

UDF invocation (nested/recursive with configurable depth)
All SQL datatypes.
**FROID UDF IMPROVEMENT STUDY**

**Table: 100k Tuples**

<table>
<thead>
<tr>
<th>Improvement Factor</th>
<th>Azure Workload #1</th>
<th>Azure Workload #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Source:** Karthik Ramachandra

---

**Azure Workload #1**
- # of Scalar UDFs: 90
- Froid Compatible: 82 (91%)

**Azure Workload #2**
- # of Scalar UDFs: 178
- Froid Compatible: 150 (85%)
Order of magnitude "dramatic" perf gains due to Froid observed by @jdanton in @SQLServer 2019 CTP2.1!

"The other feature that I refer to as simply magic...."

"The first time I tested it, I was blown away."

Source: Karthik Ramachandra

Froid UDF IMPROVEMENT STUDY

<table>
<thead>
<tr>
<th>Azure Workload #1</th>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>82 (91%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Azure Workload #2</th>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>178</td>
<td>150 (85%)</td>
</tr>
</tbody>
</table>

Source: Karthik Ramachandra
Order of magnitude "dramatic" perf gains due to Froid observed by @jdanton in @SQLServer 2019 CTP2.1!

"The other feature that I refer to as simply magic...."

"The first time I tested at a large scale, it was mind-blowing."

Quoting from the article:

"... the CPU time is 3 times lower ... and the query is more than 20x faster!"

"For those, who use scalar UDFs extensively, the new version looks like a gift from heaven. The improvement is very impressive."

Source: Karthik Ramachandra

Azure Workload #1

<table>
<thead>
<tr>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>82 (91%)</td>
</tr>
</tbody>
</table>

Azure Workload #2

<table>
<thead>
<tr>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>150 (85%)</td>
</tr>
</tbody>
</table>
Order of magnitude "dramatic" perf gain observed by @jdanton in @SQLServer.

"The other feature that I refer to as similar to what's going on is Froid compatibility.

"The first time I tested Froid compatible was really not so good."

Table: 100k Tuples

<table>
<thead>
<tr>
<th>Azure Workload #2</th>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>178</td>
<td>150 (85%)</td>
</tr>
</tbody>
</table>

Improvement Factor

Quoting from the source:

"... the CPU time is 3 times lower ... and the query is more than 20x faster!"

"For those, who use scalar UDFs extensively, the new version looks like a gift from heaven. The improvement is very impressive."

Source: Karthik Ramachandra
Rewrite UDFs into plain SQL commands.

Use recursive common table expressions (CTEs) to support iterations and other control flow concepts not supported in Froid.

Implemented as a rewrite middleware layer on top of any DBMS that supports CTEs.

→ Online Demo: https://apfel-db.cs.uni-tuebingen.de/
APFEL: UDFs-TO-CTEs OVERVIEW

Step #1 – Static Single Assignment Form
Step #2 – Administrative Normal Form
Step #3 – Mutual to Direct Recursion
Step #4 – Tail Recursion to WITH RECURSIVE
Step #5 – Run Through Query Optimizer
**STEP #1: STATIC SINGLE ASSIGNMENT**

```
CREATE FUNCTION pow(x int, n int)
RETURNS int AS
$$
DECLARE
  i int = 0;
  p int = 1;
BEGIN
  WHILE i < n LOOP
    p = p * x;
    i = i + 1;
  END LOOP;
RETURN p;
END;
$$
```

**Source:** Torsten Grust
STEP #2: ADMINISTRATIVE NORMAL FORM

\[
\text{pow}(x, n):
\begin{align*}
& i_0 \leftarrow 0; \\
& p_0 \leftarrow 0; \\
& \text{while: } i_1 \leftarrow \Phi(i_0, i_2); \\
& \quad p_1 \leftarrow \Phi(p_0, p_2); \\
& \quad \text{if } i_1 < n \text{ then } \\
& \quad \quad \text{goto loop;} \\
& \quad \text{else} \\
& \quad \quad \text{goto exit;} \\
& \text{loop: } p_2 \leftarrow p_1 \times x; \\
& \quad i_2 \leftarrow i_1 + 1; \\
& \quad \text{goto while; } \\
& \text{exit: return } p_1;
\end{align*}
\]

\[
\text{pow}(x, n) = \\
\begin{align*}
& \text{let } i_0 = 0 \text{ in} \\
& \quad \text{let } p_0 = 1 \text{ in} \\
& \quad \text{while}(i_0, p_0, x, n) \\
& \text{while}(i_1, p_1, x, n) = \\
& \begin{align*}
& \quad \text{let } t_0 = i_1 \geq n \text{ in} \\
& \quad \quad \text{if } t_0 \text{ then } p_1 \\
& \quad \quad \text{else } \text{body}(i_1, p_1, x, n) \\
& \text{body}(i_1, p_1, x, n) = \\
& \begin{align*}
& \quad \text{let } p_2 = p_1 \times x \text{ in} \\
& \quad \text{let } i_2 = i_1 + 1 \text{ in} \\
& \quad \text{while}(i_2, p_2, x, n)
\end{align*}
\end{align*}
\]

Source: Torsten Grust
**STEP #3: MUTUAL TO DIRECT RECURSION**

\[
\text{pow}(x,n) = \\
\quad \text{let } i_0 = 0 \text{ in} \\
\quad \text{let } p_0 = 1 \text{ in} \\
\quad \text{while}(i_0,p_0,x,n)
\]

\[
\text{while}(i_1,p_1,x,n) = \\
\quad \text{let } t_0 = i_1 \geq n \text{ in} \\
\quad \text{if } t_0 \text{ then } p_1 \\
\quad \text{else } \text{body}(i_1,p_1,x,n)
\]

\[
\text{body}(i_1,p_1,x,n) = \\
\quad \text{let } p_2 = p_1 \times x \text{ in} \\
\quad \text{let } i_2 = i_1 + 1 \text{ in} \\
\quad \text{while}(i_2,p_2,x,n)
\]

Source: Torsten Grust
**STEP #4: WITH RECURSIVE**

\[ \text{pow}(x,n) = \]
\[ \text{let } i_0 = 0 \text{ in} \]
\[ \text{let } p_0 = 1 \text{ in} \]
\[ \text{run}(i_0,p_0,x,n) \]

\[ \text{run}(i_1,p_1,x,n) = \]
\[ \text{let } t_0 = i_1 \geq n \text{ in} \]
\[ \text{if } t_0 \text{ then } p_1 \]
\[ \text{else} \]
\[ \text{let } p_2 = p_1 \times x \text{ in} \]
\[ \text{let } i_2 = i_1 + 1 \text{ in} \]
\[ \text{run}(i_2,p_2,x,n) \]

---

**WITH RECURSIVE**

\[ \text{run}("call?",i_1,p_1,x,n,result) \text{ AS (} \]
\[ \text{SELECT } \text{true},0,1,x,n,\text{NULL} \]
\[ \text{UNION ALL} \]
\[ \text{SELECT } \text{iter.}* \text{ FROM run, LATERAL (} \]
\[ \text{SELECT } \text{false},0,0,0,0,p_1 \]
\[ \text{WHERE } i_1 \geq n \]
\[ \text{UNION ALL} \]
\[ \text{SELECT } \text{true},i_1+1,p_1\times x,x,n,0 \]
\[ \text{WHERE } i_1 < n \]
\[ ) \text{ AS } \text{iter}("call?",i_1,p_1,x,n,result) \]
\[ ) \]
\[ \text{SELECT } * \text{ FROM run}; \]

Source: Torsten Grust
STEP #4: WITH RECURSIVE

\[
opow(x, n) = \\
\begin{align*}
&\text{let } i_0 = 0 \text{ in} \\
&\text{let } p_0 = 1 \text{ in} \\
&\text{run}(i_0, p_0, x, n) \\
\end{align*}
\]

\[
\text{run}(i_1, p_1, x, n) = \\
\begin{align*}
&\text{let } t_0 = i_1 \geq n \text{ in} \\
&\text{if } t_0 \text{ then } p_1 \\
&\text{else} \\
&\text{let } p_2 = p_1 \times x \text{ in} \\
&\text{let } i_2 = i_1 + 1 \text{ in} \\
&\text{run}(i_2, p_2, x, n) \\
\end{align*}
\]

WITH RECURSIVE
\[
\text{run}("call?", i_1, p_1, x, n, \text{result}) \ AS ( \\
\text{SELECT true, 0, 1, x, n, NULL} \\
\text{UNION ALL} \\
\text{SELECT iter.* FROM run, LATERAL (} \\
\text{SELECT false, 0, 0, 0, 0, p_1} \\
\text{WHERE i_1 \geq n} \\
\text{UNION ALL} \\
\text{SELECT true, i_1+1, p_1*x, x, n, 0} \\
\text{WHERE i_1 < n} \\
\text{)} \ AS iter("call?", i_1, p_1, x, n, \text{result}) \text{ WHERE run."call?"} \\
\text{)} \\
\text{SELECT * FROM run;}
\]
**STEP #4: WITH RECURSIVE**

1. $\text{pow}(x, n) =$
   - let $i_0 = 0$ in
   - let $p_0 = 1$ in
     - run($i_0, p_0, x, n$)

2. run($i_1, p_1, x, n$) =
   - let $t_0 = i_1 \geq n$ in
     - if $t_0$ then $p_1$
     - else

3. let $p_2 = p_1 \times x$ in
   - let $i_2 = i_1 + 1$ in
     - run($i_2, p_2, x, n$)

**WITH RECURSIVE**

run("call?", i1, p1, x, n, result) AS ( 

SELECT true, 0, 1, x, n, NULL
UNION ALL
SELECT iter.* FROM run, LATERAL ( 
SELECT false, 0, 0, 0, 0, p1
WHERE i1 >= n
UNION ALL
SELECT true, i1+1, p1*x, x, n, 0
WHERE i1 < n
) AS iter("call?", i1, p1, x, n, result)
WHERE run."call?"
)

SELECT * FROM run;

Source: Torsten Grust
UDFs-TO-CTEs EVALUATION

POW UDF on Postgres v11.3

Source: Torsten Grust
UDF BATCHING

Transform UDF statements into UPDATE queries that operate on a temporary table representing the state of variables in the UDF.

→ Each tuple in the state table corresponds to one input tuple to the UDF.

This method is suitable for DBMSs that are unable to decorrelate any possible subquery.
CREATE FUNCTION getManufact(item_id INT)
RETURNS CHAR(50) AS $$
DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
BEGIN
  man := ''; 
  cnt1 := (SELECT COUNT(*) 
            FROM store_sales_history, date_dim 
            WHERE ss_item_sk = item_id 
            AND d_date_sk = ss_sold_date_sk 
            AND d_year = 2023);
  cnt2 := (SELECT COUNT(*) 
            FROM catalog_sales_history, date_dim 
            WHERE cs_item_sk = item_id 
            AND d_date_sk = cs_sold_date_sk 
            AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0) 
    THEN man := (SELECT i_manufact FROM item 
                  WHERE i_item_sk = item_id);
    ELSE man := 'outdated item';
    END IF;
RETURN man;
END $$ LANGUAGE PLPGSQL;
CREATE FUNCTION getManufact(item_id INT) 
RETURNS CHAR(50) AS $$
DECLARE 
man CHAR(50); cnt1 INT; cnt2 INT;
BEGIN 
man := ''; 
cnt1 := (SELECT COUNT(*)
FROM store_sales_history, date_dim
WHERE ss_item_sk = item_id
AND d_date_sk = ss_sold_date_sk
AND d_year = 2023);
cnt2 := (SELECT COUNT(*)
FROM catalog_sales_history, date_dim
WHERE cs_item_sk = item_id
AND d_date_sk = cs_sold_date_sk
AND d_year = 2023);
IF (cnt1 > 0 AND cnt2 > 0)
THEN man := (SELECT i_manufact FROM item
WHERE i_item_sk = item_id);
ELSE man := 'outdated item';
END IF;
RETURN man;
END $$ LANGUAGE PLPGSQL;

Source: Kai Franz
CREATE FUNCTION getManufact(item_id INT) RETURNS CHAR(50) AS $$
DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
BEGIN
  man := '';
  cnt1 := (SELECT COUNT(*)
            FROM store_sales_history, date_dim
            WHERE ss_item_sk = item_id
            AND d_date_sk = ss_sold_date_sk
            AND d_year = 2023);
  cnt2 := (SELECT COUNT(*)
            FROM catalog_sales_history, date_dim
            WHERE cs_item_sk = item_id
            AND d_date_sk = cs_sold_date_sk
            AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0)
     THEN man := (SELECT i_manufact
                   FROM item
                   WHERE i_item_sk = item_id);
  ELSE man := 'outdated item';
  END IF;
  RETURN man;
END $$ LANGUAGE PLPGSQL;

SELECT ws_item_sk FROM (SELECT ws_item_sk, COUNT(*) AS cnt
                         FROM web_sales
                         GROUP BY ws_item_sk
                         ORDER BY cnt DESC, ws_item_sk
                         LIMIT 25000) AS t1
WHERE getManufact(ws_item_sk) = 'CompanyX';
CREATE FUNCTION getManufact(item_id INT) 
RETURNS CHAR(50) AS $$
DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
BEGIN
  man := '';
  cnt1 := (SELECT COUNT(*)
            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(*)
            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)
    THEN man := (SELECT i_manufact FROM item
                     WHERE i_item_sk = item_id);
    ELSE man := 'outdated item';
  END IF;
  RETURN man;
END $$ LANGUAGE PLPGSQL;

SELECT ws_item_sk
FROM (
  SELECT ws_item_sk,
         COUNT(*) AS cnt
  FROM web_sales
  GROUP BY ws_item_sk
  ORDER BY cnt DESC, ws_item_sk
  LIMIT 25000) AS t1
WHERE getManufact(ws_item_sk) = 'CompanyX';
CREATE FUNCTION getManufact(item_id INT) RETURNS CHAR(50) AS $$
DECLARE 
  man CHAR(50); cnt1 INT; cnt2 INT;
BEGIN 
  man := '';
  cnt1 := (SELECT COUNT(*) FROM store_sales_history, date_dim 
           WHERE ss_item_sk = item_id
           AND d_date_sk = ss_sold_date_sk
           AND d_year = 2023);
  cnt2 := (SELECT COUNT(*) FROM catalog_sales_history, date_dim 
           WHERE cs_item_sk = item_id
           AND d_date_sk = cs_sold_date_sk
           AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0) THEN
    man := (SELECT i_manufact FROM item 
            WHERE i_item_sk = item_id);
  ELSE
    man := 'outdated item';
  END IF;
END $$ LANGUAGE PLPGSQL;

SELECT ws_item_sk FROM 
  (SELECT ws_item_sk, COUNT(*) AS cnt
   FROM web_sales
   GROUP BY ws_item_sk
   ORDER BY cnt DESC, ws_item_sk
   LIMIT 25000) AS t1
WHERE getManufact(ws_item_sk) = 'CompanyX';
Microsoft team published an analysis of real world UDFs, TVFs, Triggers and Stored Procedures.

Also released an open-source benchmark based on their analysis called **SQL ProcBench**.

→ Authors argue that ProcBench faithfully represents real world workloads
SCALAR UDFS IN PROCBENCH

**UDFs with No Parameters**

```
SELECT maxReturnReasonWeb();
```

```
CREATE FUNCTION maxReturnReasonWeb()
RETURNS char(100) AS
BEGIN
    DECLARE @reason_desc char(100);
    SELECT @reason_desc
    FROM ...;
    RETURN @reason_desc;
END
```

UDF invoked once
No substantial performance advantage with UDF Inlining

Source: Sam Arch
SCALAR UDFS IN THE PROCBENCH

**UDFs with Parameters**

CREATE FUNCTION `cust_level`(@ckey int) 
RETURNS char(10) AS
BEGIN
    DECLARE @total float;
    DECLARE @level char(10);

    SELECT @total = SUM(o_totalprice)
    FROM orders WHERE o_custkey=@ckey;

    IF (@total > 1000000)
        SET @level = 'Platinum';
    ELSE
        SET @level = 'Regular';

    RETURN @level;
END

UDF invoked per customer
Implicit join between tables
Huge performance win with inlining by “decorrelating” the subquery

SELECT `cust_level`(customer_id)
FROM customer;
## UDF Batching vs. Inlining

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>18</th>
<th>20a_q1</th>
<th>20a_q2</th>
<th>20b_q1</th>
<th>20b_q2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL Server</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inlined</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batched</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Oracle</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inlined</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batched</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>DuckDB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inlined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batched</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>PostgreSQL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inlined</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Batched</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**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.
## UDF Batching vs. Inlining

<table>
<thead>
<tr>
<th>Method</th>
<th>SQL Server</th>
<th>Oracle</th>
<th>DuckDB</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inlined</td>
<td>Batched</td>
<td>Inlined</td>
<td>Batched</td>
</tr>
<tr>
<td>UDFs</td>
<td>1 5 6 7 12 13 15 17 18 20a_q1 20a_q2 20b_q1 20b_q2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL Server</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DuckDB</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batched</td>
<td>( )</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batched</td>
<td>x x x x x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.
# UDF Batching vs. Inlining

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>18</th>
<th>20a_q1</th>
<th>20a_q2</th>
<th>20b_q1</th>
<th>20b_q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server</td>
<td>Inlined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Inlined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DuckDB</td>
<td>Inlined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Inlined</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Subquery Dependencies

The table above shows the dependencies between UDFs in various databases. The green checkmark indicates that the UDFs are inlined, while the red X indicates batching. The black pen indicates that some, but not all, UDFs are inlined.

---

**Implementing & Flattening Nested LATERAL joins in DuckDB**

Sam Arch, Mayank Baranwal, Arham Chopra
Add support for nested laterals #7528

by Anyherin

* duplicated

- 2023-05-22 16:00

12 commits

18 conversations

Contributor

This PR adds support for nested LATERAL joins (arbitrary nesting of subqueries and LATERAL joins) to DuckDB. In the current version of DuckDB, the following example from PR #5373 will produce a binder error:

```
SELECT * FROM (SELECT x) t(x), (SELECT k,x t(k,l)) t2(l);
```

# Binder Error: Nested lateral joins are not supported yet

However, after this PR, DuckDB produces the correct result:

```
SELECT * FROM (SELECT x) t(x), (SELECT k,x t(k,l)) t2(l);
```

Further, after this PR, queries with correlations across LATERALS and subqueries also produce the correct result:
# UDF Batching vs. Inlining

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>18</th>
<th>20a_q1</th>
<th>20a_q2</th>
<th>20b_q1</th>
<th>20b_q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inlined</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Batched</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Oracle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Inlined</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Batched</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>DuckDB</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Inlined</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Batched</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Inlined</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Batched</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**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.
**DECORRELATION OF SUBQUERIES (MSSQL)**

**Algebraic rewrite rules for APPLY**

\[
R \ A^\otimes E = R \otimes_{\text{true}} E, \quad (1)
\]

if no parameters in \( E \) resolved from \( R \)

\[
R \ A^\otimes (\sigma_p E) = R \otimes_p E, \quad (2)
\]

if no parameters in \( E \) resolved from \( R \)

\[
R \ A^\times (\sigma_p E) = \sigma_p(R A^\times E) \quad (3)
\]

\[
R \ A^\times (\pi_v E) = \pi_v \cup \text{columns}(R)(R A^\times E) \quad (4)
\]

\[
R \ A^\times (E_1 \cup E_2) = (R A^\times E_1) \cup (R A^\times E_2) \quad (5)
\]

\[
R \ A^\times (E_1 - E_2) = (R A^\times E_1) - (R A^\times E_2) \quad (6)
\]

\[
R \ A^\times (E_1 \times E_2) = (R A^\times E_1) \Join_{R,\text{key}} (R A^\times E_2) \quad (7)
\]

\[
R \ A^\times (G_{A,F} E) = G_{A \cup \text{columns}(R), F}(R A^\times E) \quad (8)
\]

\[
R \ A^\times (G_F^{1} E) = G_{\text{columns}(R), F'}(R A^{\text{LOJ}} E) \quad (9)
\]

 Execute the rewrite rules where applicable

Some rewrites may require duplicating subexpressions in the query plan tree (and are cost-based decisions)
**Dependent Join Operator**

\[
\sigma_{e.grade=m}
\]

\[
\Join_{s.id=e.sid}
\]

\[
\Gamma_0;m:\min(e2.grade)
\]

\[
\text{students } s \quad \text{exams } e \quad \sigma_{s.id=e2.sid}
\]

\[
\text{exams } e2
\]

Introduces a new “Dependent Join” operator into the Query Plan DAG

Systematically decorrelates any subquery
PARTING THOUGHTS

This is huge. You rarely get 500x speed up without either switching to a new DBMS or rewriting your application.

But the DBMS **must** support German-style (aka HyPer) sub-query decorrelation.

Another optimization approach is to compile the UDF into machine code.

→ This does **not** solve the optimizer's cost model problem.
Database Networking Protocols

And a little bit about kernel bypass methods...