ADVANCED
DATABASE
SYSTEMS

User-Defined Functions

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

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

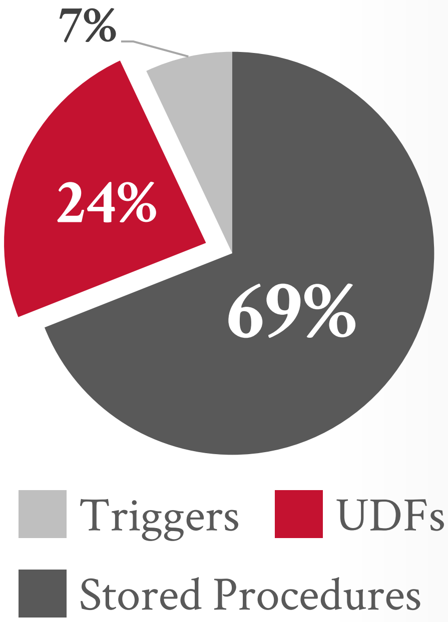
User-Defined Functions (UDFs)

Stored Procedures

Triggers

User-Defined Types (UDTs)

User-Defined Aggregates (UDAs)



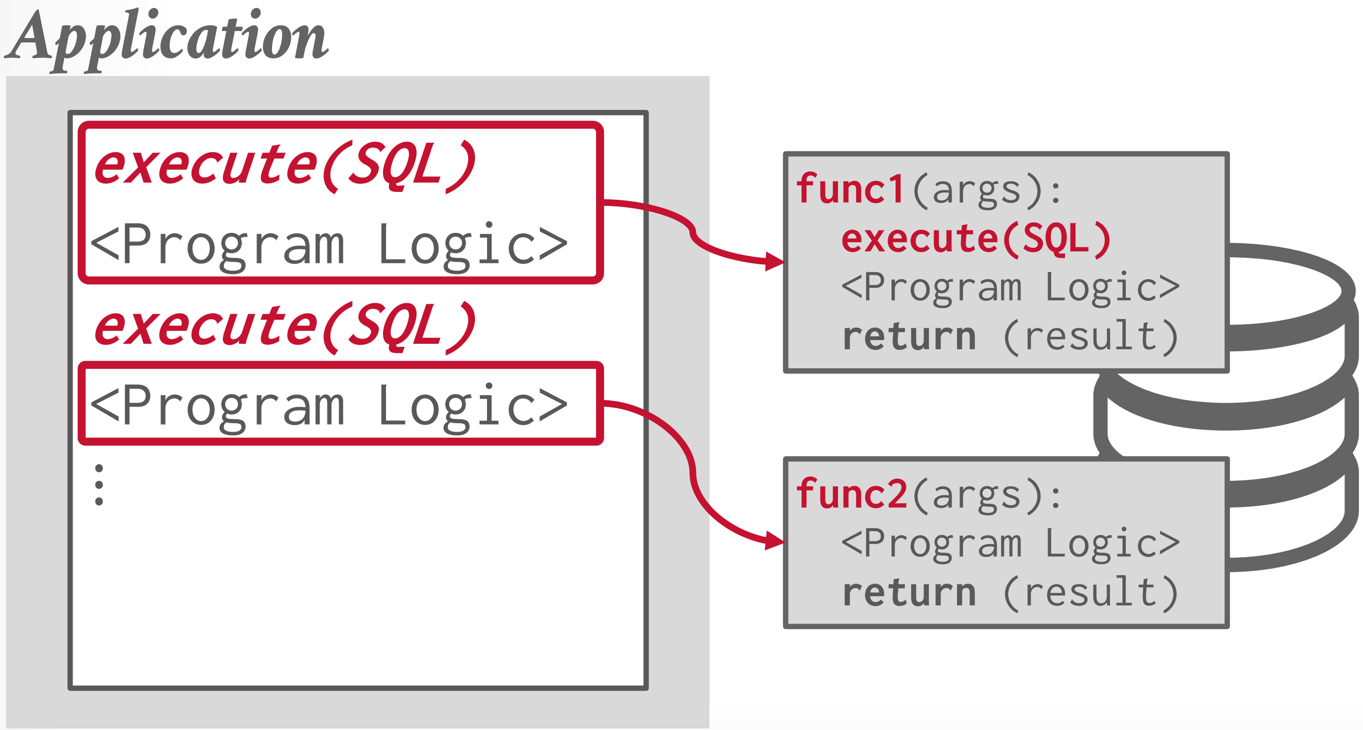
PROCEDURAL EXTENSIONS OF SQL:
UNDERSTANDING THEIR USAGE IN THE WILD
VLDB 2021

USER-DEFINED FUNCTIONS

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)

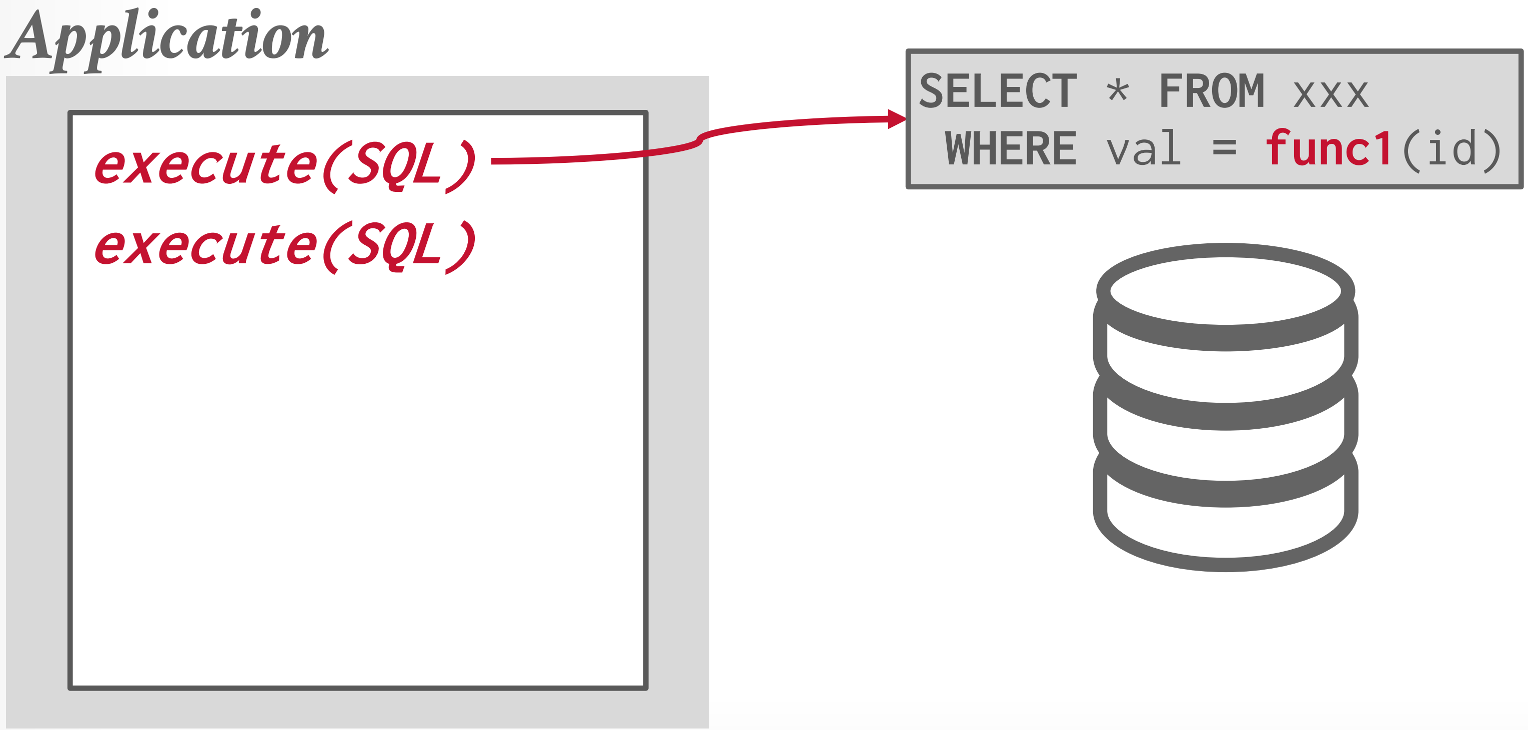
USER-DEFINED FUNCTIONS



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

```
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

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 $$
  SELECT * FROM foo WHERE foo.id = $1;
$$;
```

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.

```
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.

```
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

UDF: EXTERNAL PROGRAMMING LANGUAGE

```
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
```

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

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

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.

UDF DISADVANTAGES (1)

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)

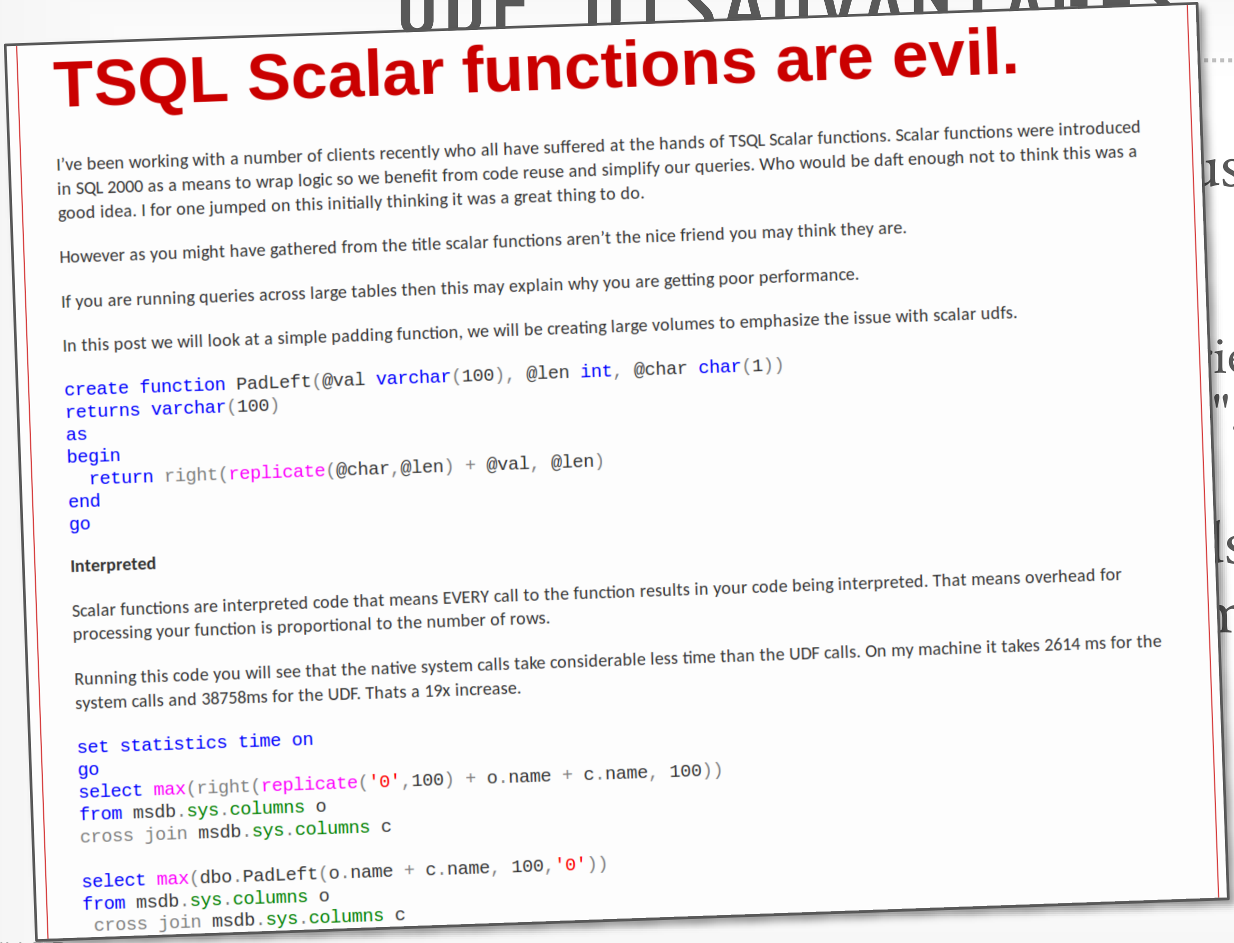
Complex UDFs in **SELECT** / **WHERE** clauses force the DBMS to execute iteratively.

→ 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)



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.

```
create function PadLeft(@val varchar(100), @len int, @char char(1))
returns varchar(100)
as
begin
  return right(replicate(@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 considerable less time than the UDF calls. On my machine it takes 2614 ms for the system calls and 38758ms for the UDF. Thats a 19x increase.

```
set statistics time on
go
select max(right(replicate('0',100) + o.name + c.name, 100))
from msdb.sys.columns o
cross join msdb.sys.columns c

select max(dbo.PadLeft(o.name + c.name, 100,'0'))
from msdb.sys.columns o
 cross join msdb.sys.columns c
```

uses force the

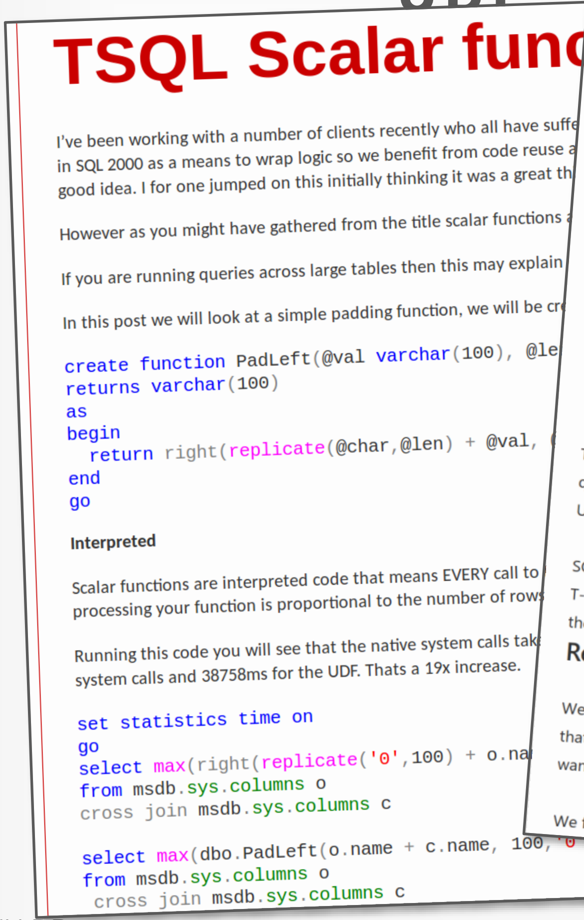
ies due to

".

ls in the

n cross-

UDF DISADVANTAGES (2)



TSQL Scalar func

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

However as you might have gathered from the title scalar functions a

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 cr

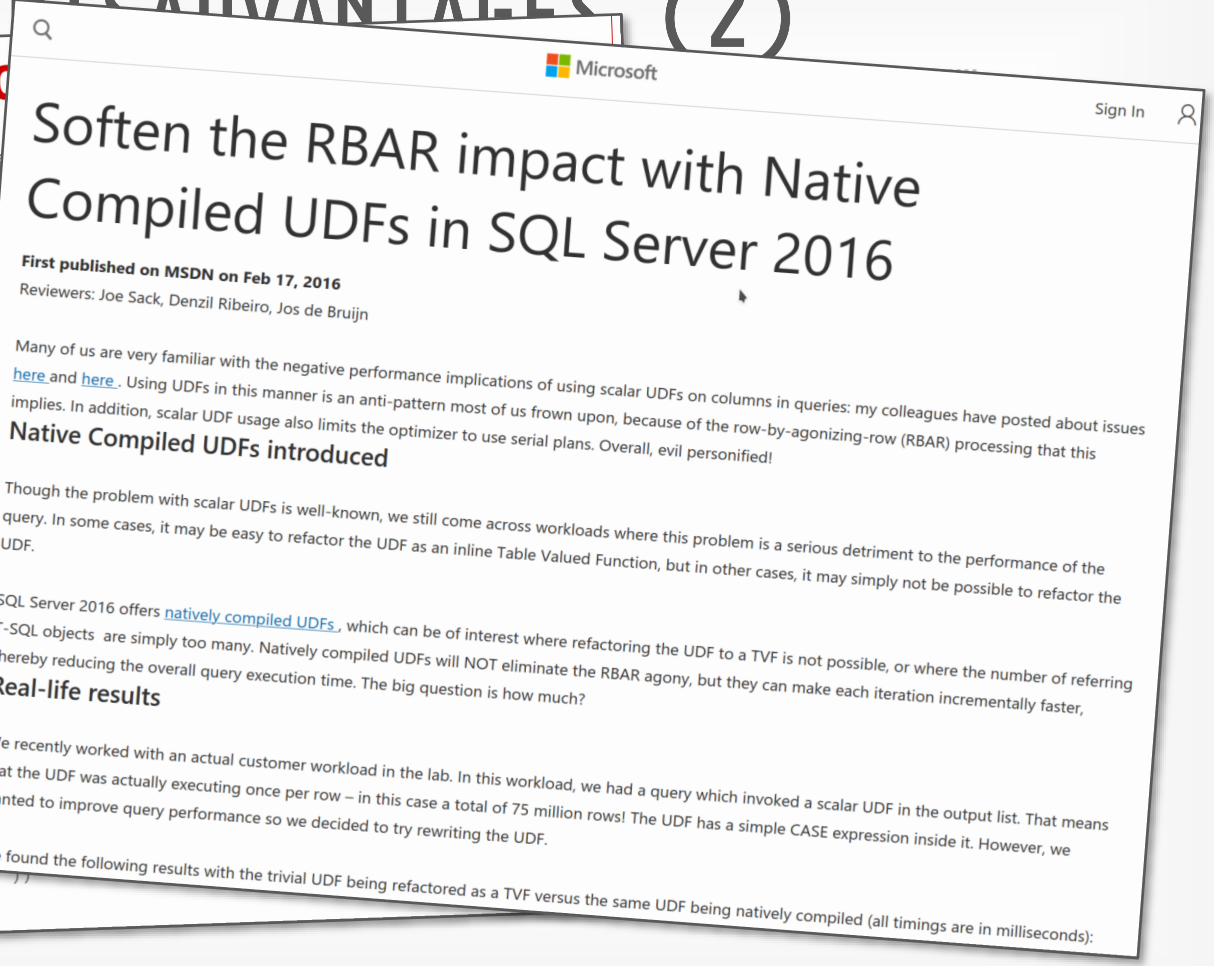
```
create function PadLeft(@val varchar(100), @le
returns varchar(100)
as
begin
  return right(replicate(@char,@len) + @val, (
end
go
```

Interpreted

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

Running this code you will see that the native system calls tak
system calls and 38758ms for the UDF. Thats a 19x increase.

```
set statistics time on
go
select max(right(replicate('0',100) + o.na
from msdb.sys.columns o
cross join msdb.sys.columns c

select max(dbo.PadLeft(o.name + c.name, 100, '0'))
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

Reviewers: Joe Sack, Denzil Ribeiro, Jos de Bruijn

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 frown upon, because of the row-by-agonizing-row (RBAR) processing that this implies. In addition, scalar UDF usage 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, thereby reducing the overall query execution time. The big question is how much?

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 that 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):

UDF PERFORMANCE

Microsoft SQL Server

```
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
```

TPC-H Q12 using a UDF (SF=1).

→ **Original Query:** 0.8 sec

→ **Query + UDF:** 13 hr 30 min

```
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
```

Source: Karthik Ramachandra

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.

FROID: OPTIMIZATION OF IMPERATIVE PROGRAMS
IN A RELATIONAL DATABASE
VLDB 2017

SUB-QUERIES

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…

SUB-QUERIES: REWRITE

```
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;
```

```
SELECT user_id FROM orders
 GROUP BY user_id
HAVING COUNT(*) >= 2
 ORDER BY user_id ASC LIMIT 1;
```

Example: Retrieve the first user that has made at least two purchases.

LATERAL JOIN

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.

LATERAL JOIN: EXAMPLE

```
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.

Source: Krzysztof Kempiński

LATERAL JOIN: EXAMPLE

```
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.

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

STEP #2: BREAK INTO REGIONS

```
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
```

①

```
  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
```

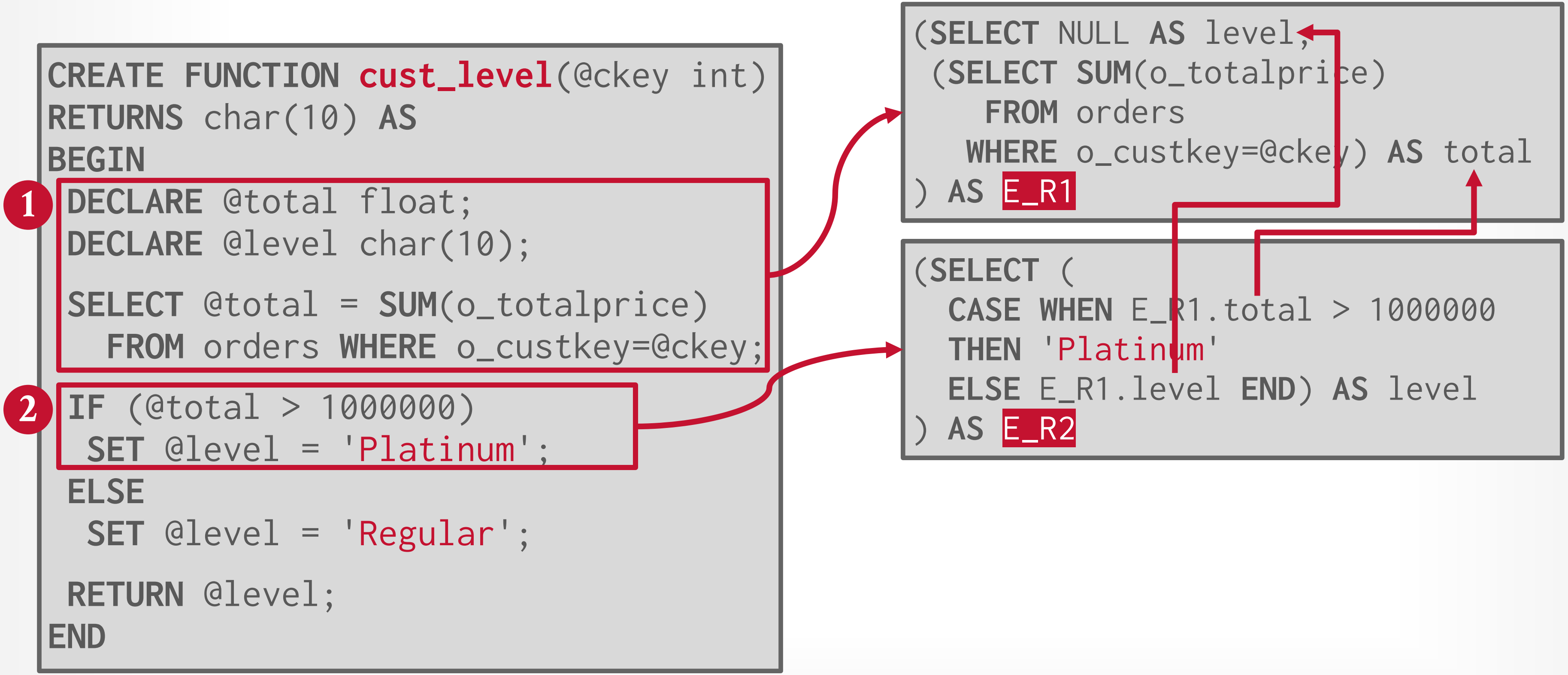
STEP #2: BREAK INTO REGIONS

```
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
```

①

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

STEP #2: BREAK INTO REGIONS

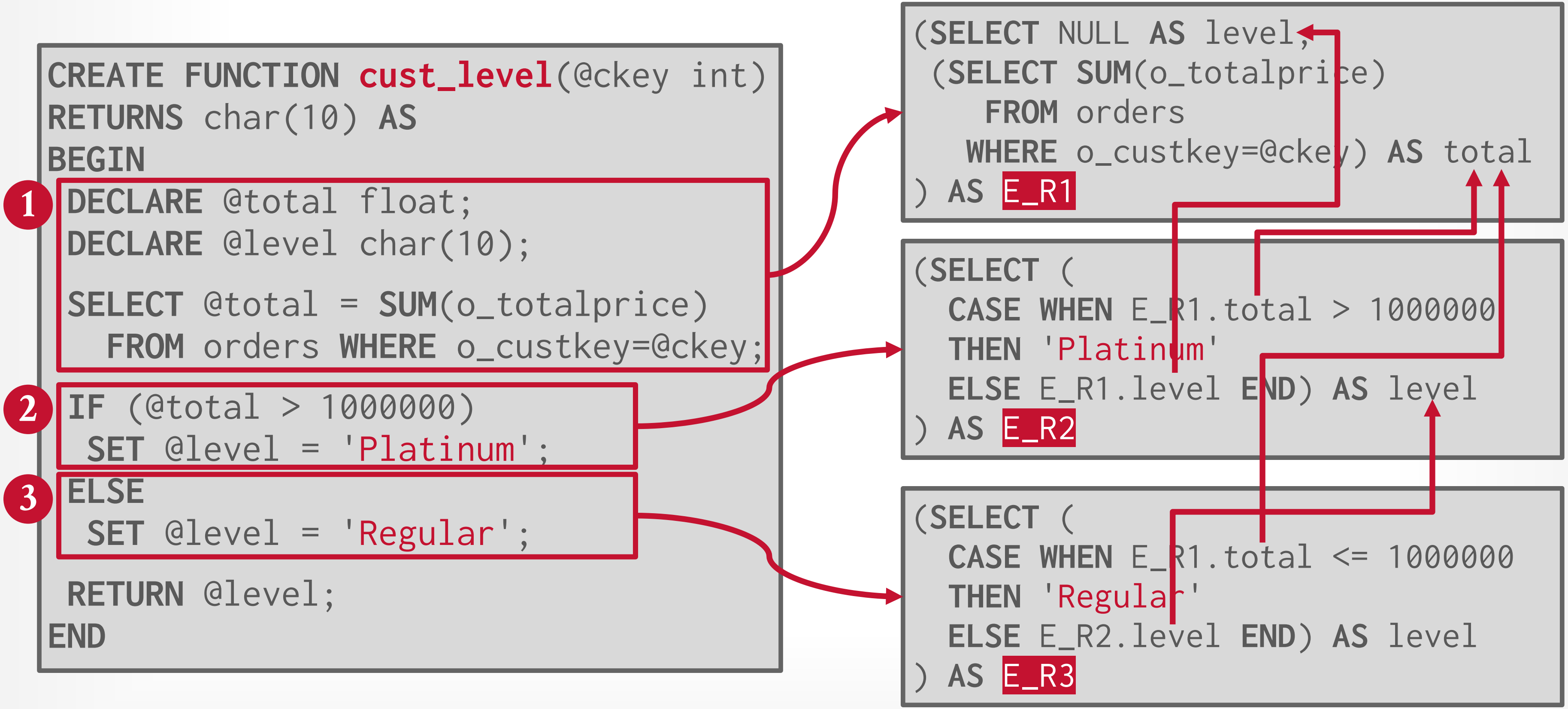


```
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
 DECLARE @total float;
 DECLARE @level char(10);

 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
```

STEP #2: BREAK INTO REGIONS



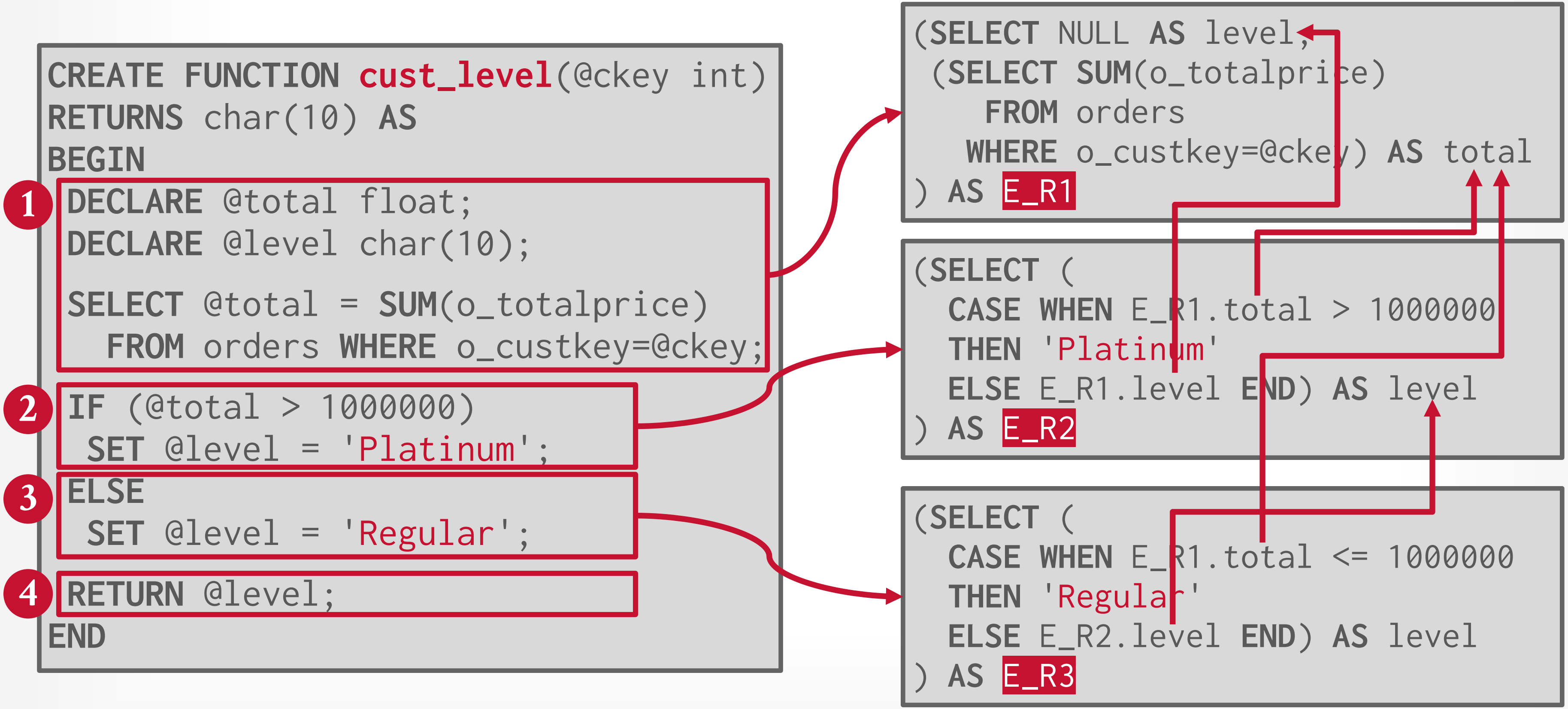
```
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
1  DECLARE @total float;
   DECLARE @level char(10);

   SELECT @total = SUM(o_totalprice)
     FROM orders WHERE o_custkey=@ckey;
2  IF (@total > 1000000)
    SET @level = 'Platinum';
3  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 #2: BREAK INTO REGIONS



```
CREATE FUNCTION cust_level(@ckey int)
RETURNS char(10) AS
BEGIN
①  DECLARE @total float;
   DECLARE @level char(10);

   SELECT @total = SUM(o_totalprice)
     FROM orders WHERE o_custkey=@ckey;
②  IF (@total > 1000000)
     SET @level = 'Platinum';
③  ELSE
     SET @level = 'Regular';
④  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
```

4

```
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
```

```
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_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;
```

BONUS OPTIMIZATIONS

```
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);
```

BONUS OPTIMIZATIONS

```
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
```

Inline

```
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 returnVal) E_R2
```

Dynamic Slicing

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

```
SELECT returnVal FROM
 (SELECT 'high' AS val)
 AS E_R1
 OUTER APPLY
 (SELECT DT1.val +
         ' value'
         AS returnVal)
 AS E_R2
```

Constant Propagation & Folding

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

```
SELECT returnVal FROM
 (SELECT 'high value'
         AS returnVal)
 AS E_R1
```

Dead Code Elimination

```
BEGIN
 RETURN 'high value';
END
```

```
SELECT 'high value';
```

SUPPORTED OPERATIONS (2019)

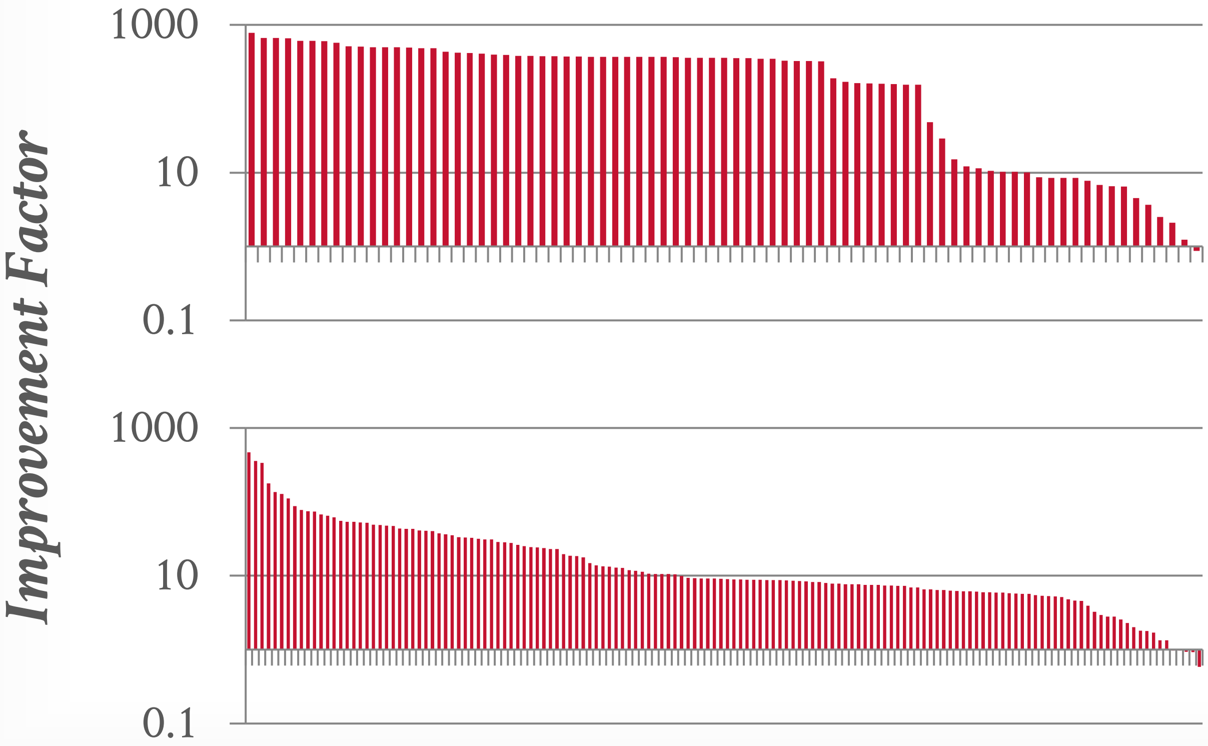
T-SQL Syntax:
- → **DECLARE**, **SET** (variable declaration, assignment)
- → **SELECT** (SQL query, assignment)
- → **IF** / **ELSE** / **ELSE IF** (arbitrary nesting)
- → **RETURN** (multiple occurrences)
- → **EXISTS**, **NOT EXISTS**, **ISNULL**, **IN**, … (Other relational algebra operations)

UDF invocation (nested/recursive with configurable depth)

All SQL datatypes.

FROID UDF IMPROVEMENT STUDY

Table: 100k Tuples



Azure Workload #1

# of Scalar UDFs 90

Froid Compatible 82 **(91%)**

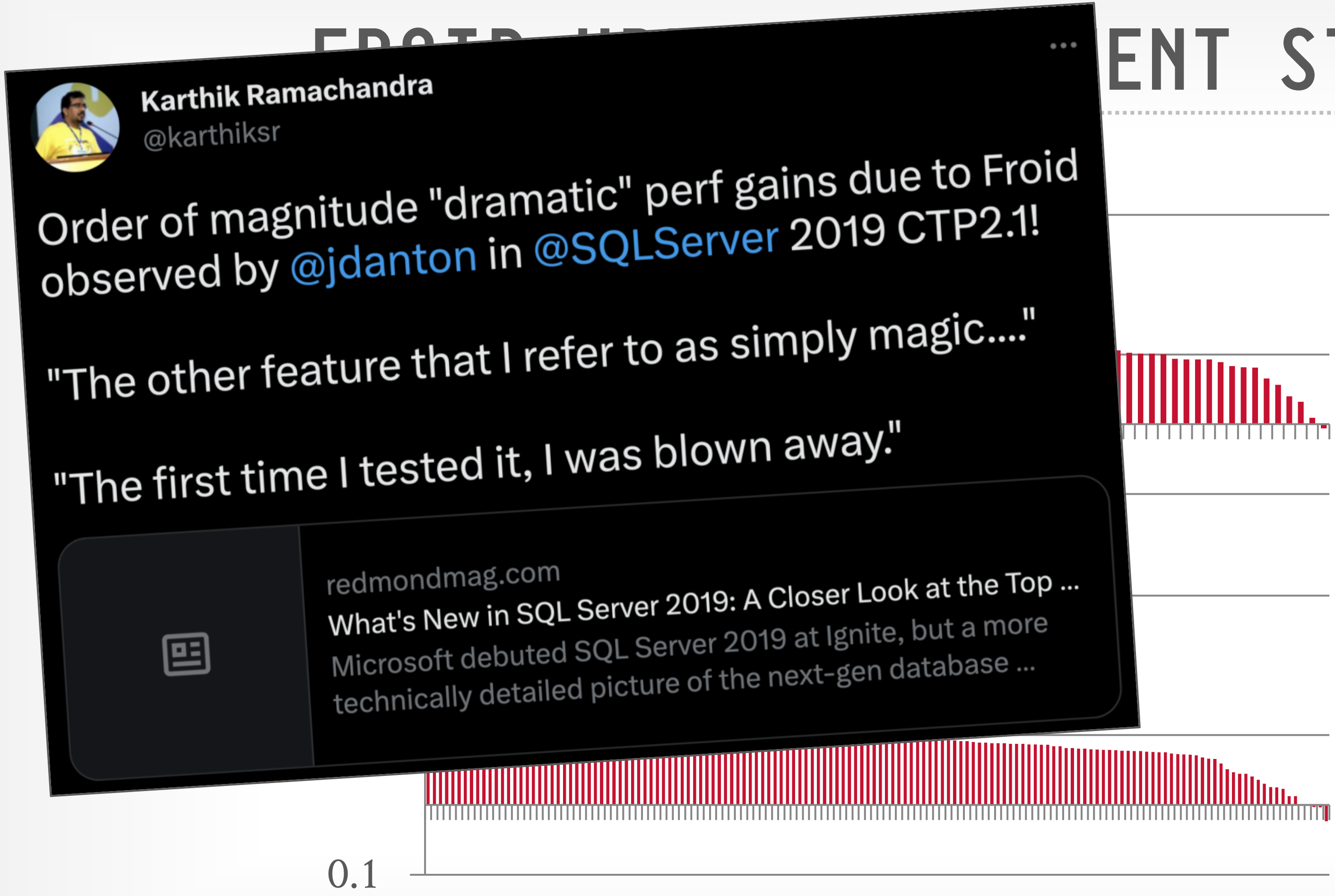
Azure Workload #2

# of Scalar UDFs 178

Froid Compatible 150 **(85%)**

Source: Karthik Ramachandra

ENT STUDY



Karthik Ramachandra
@karthiksr

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."

redmondmag.com
What's New in SQL Server 2019: A Closer Look at the Top ...
Microsoft debuted SQL Server 2019 at Ignite, but a more technically detailed picture of the next-gen database ...

Azure Workload #1

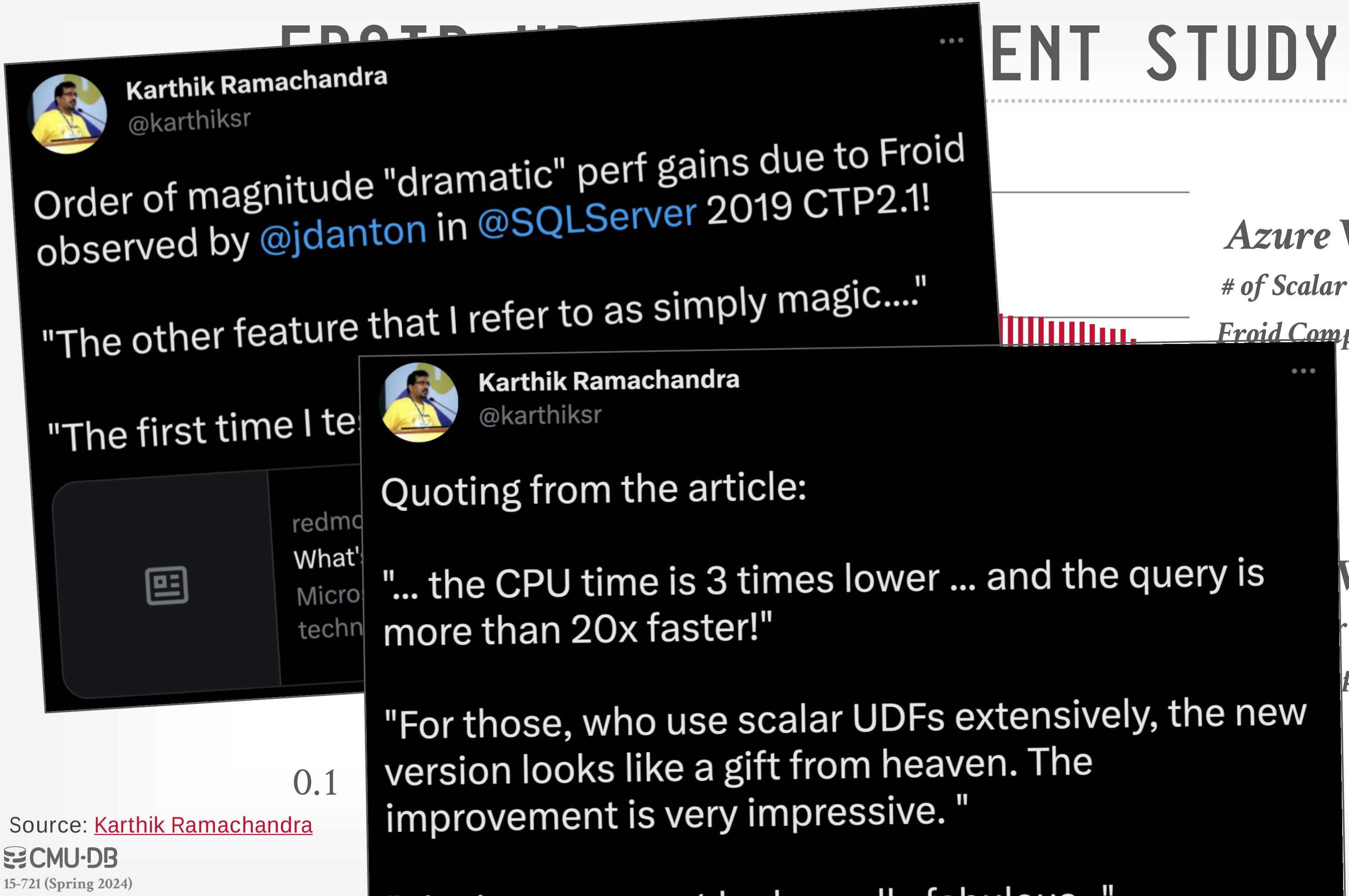
# of Scalar UDFs	90
Froid Compatible	82 **(91%)**

Azure Workload #2

# of Scalar UDFs	178
Froid Compatible	150 **(85%)**

Source: Karthik Ramachandra

ENT STUDY



Karthik Ramachandra
@karthiksr

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 te

redmo
What'
Micro
techn

Karthik Ramachandra
@karthiksr

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. "

Azure Workload #1

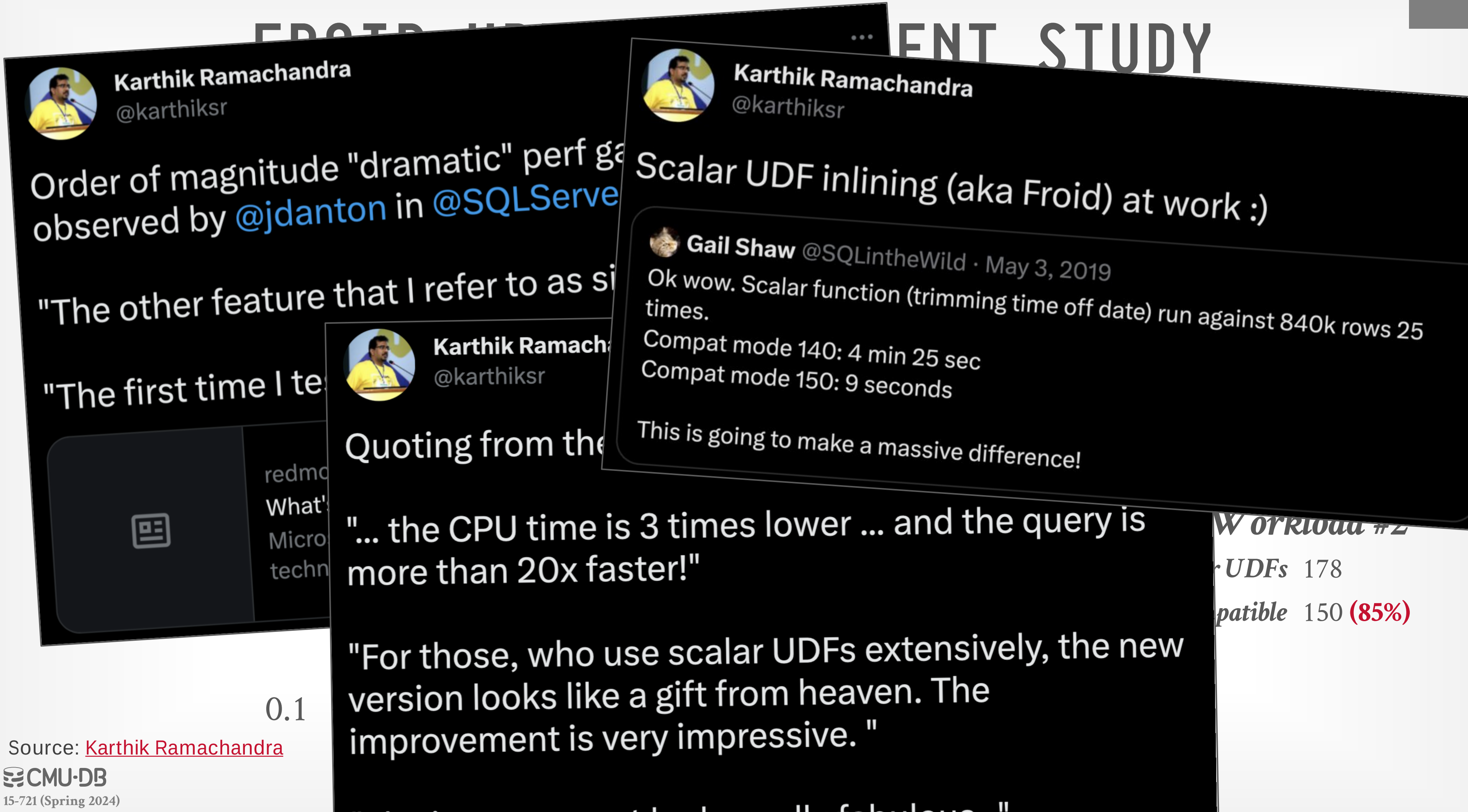
# of Scalar UDFs	90
Froid Compatible	82 **(91%)**

Workload #2

r UDFs	178
patible	150 **(85%)**

Source: Karthik Ramachandra

ENT STUDY



Karthik Ramachandra
@karthiksr

Order of magnitude "dramatic" perf ga
observed by @jdanton in @SQLServe

"The other feature that I refer to as si

"The first time I te

redmo
What'
Micro
techn

Karthik Ramachandra
@karthiksr

Scalar UDF inlining (aka Froid) at work :)

Gail Shaw @SQLintheWild · May 3, 2019
Ok wow. Scalar function (trimming time off date) run against 840k rows 25 times.
Compat mode 140: 4 min 25 sec
Compat mode 150: 9 seconds

This is going to make a massive difference!

Karthik Ramacha
@karthiksr

Quoting from the

"... 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. "

Workload #2

UDFs 178

patible 150 **(85%)**

0.1

Source: Karthik Ramachandra

APFEL: UDFs-TO-CTEs

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/

COMPILING PL/SQL AWAY
CIDR 2020

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
$$
 DECLARE
  i int = 0;
  p int = 1;
 BEGIN
  WHILE i < n LOOP
   p = p * x;
   i = i + 1;
  END LOOP;
  RETURN p;
 END;
$$
```

```
pow(x,n):
         $i_0 \leftarrow 0$;
         $p_0 \leftarrow 0$;
 while:  $i_1 \leftarrow \Phi(i_0, i_2)$;
         $p_1 \leftarrow \Phi(p_0, p_2)$;
         if $i_1 < n$ then
           goto loop;
         else
           goto exit;
  loop:  $p_2 \leftarrow p_1 * x$;
         $i_2 \leftarrow i_1 + 1$;
         goto while;
  exit:  return $p_1$;
```

Source: Torsten Grust

STEP #2: ADMINISTRATIVE NORMAL FORM

```
pow(x,n):
        $i_0$ ← 0;
        $p_0$ ← 0;
 while: $i_1$ ← Φ($i_0$,$i_2$);
        $p_1$ ← Φ($p_0$,$p_2$);
        if $i_1$ < n then
          goto loop;
        else
          goto exit;
  loop: $p_2$ ← $p_1$ * x;
        $i_2$ ← $i_1$ + 1;
        goto while;
  exit: return $p_1$;
```

```
pow(x,n) =
  let $i_0$ = 0 in
   let $p_0$ = 1 in
     while($i_0$,$p_0$,x,n)

while($i_1$,$p_1$,x,n) =
  let $t_0$ = $i_1$ >= n in
   if $t_0$ then $p_1$
   else body($i_1$,$p_1$,x,n)

body($i_1$,$p_1$,x,n) =
  let $p_2$ = $p_1$ * x in
   let $i_2$ = $i_1$ + 1 in
    while($i_2$,$p_2$,x,n)
```

Source: Torsten Grust

STEP #3: MUTUAL TO DIRECT RECURSION

```
pow(x,n) =
  let i_0 = 0 in
   let p_0 = 1 in
     while(i_0,p_0,x,n)

while(i_1,p_1,x,n) =
  let t_0 = i_1 >= n in
   if t_0 then p_1
   else body(i_1,p_1,x,n)

body(i_1,p_1,x,n) =
  let p_2 = p_1 * x in
   let i_2 = i_1 + 1 in
    while(i_2,p_2,x,n)
```

```
pow(x,n) =
  let i_0 = 0 in
   let p_0 = 1 in
     run(i_0,p_0,x,n)

run(i_1,p_1,x,n) =
  let t_0 = i_1 >= n in
   if t_0 then p_1
   else
    let p_2 = p_1 * x in
     let i_2 = i_1 + 1 in
      run(i_2,p_2,x,n)
```

Source: Torsten Grust

STEP #4: WITH RECURSIVE

```
pow(x,n) =
  let $i_0$ = 0 in
   let $p_0$ = 1 in
     run($i_0$,$p_0$,x,n)

run($i_1$,$p_1$,x,n) =
  let $t_0$ = $i_1$ >= n in
   if $t_0$ then $p_1$
   else
    let $p_2$ = $p_1$ * 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

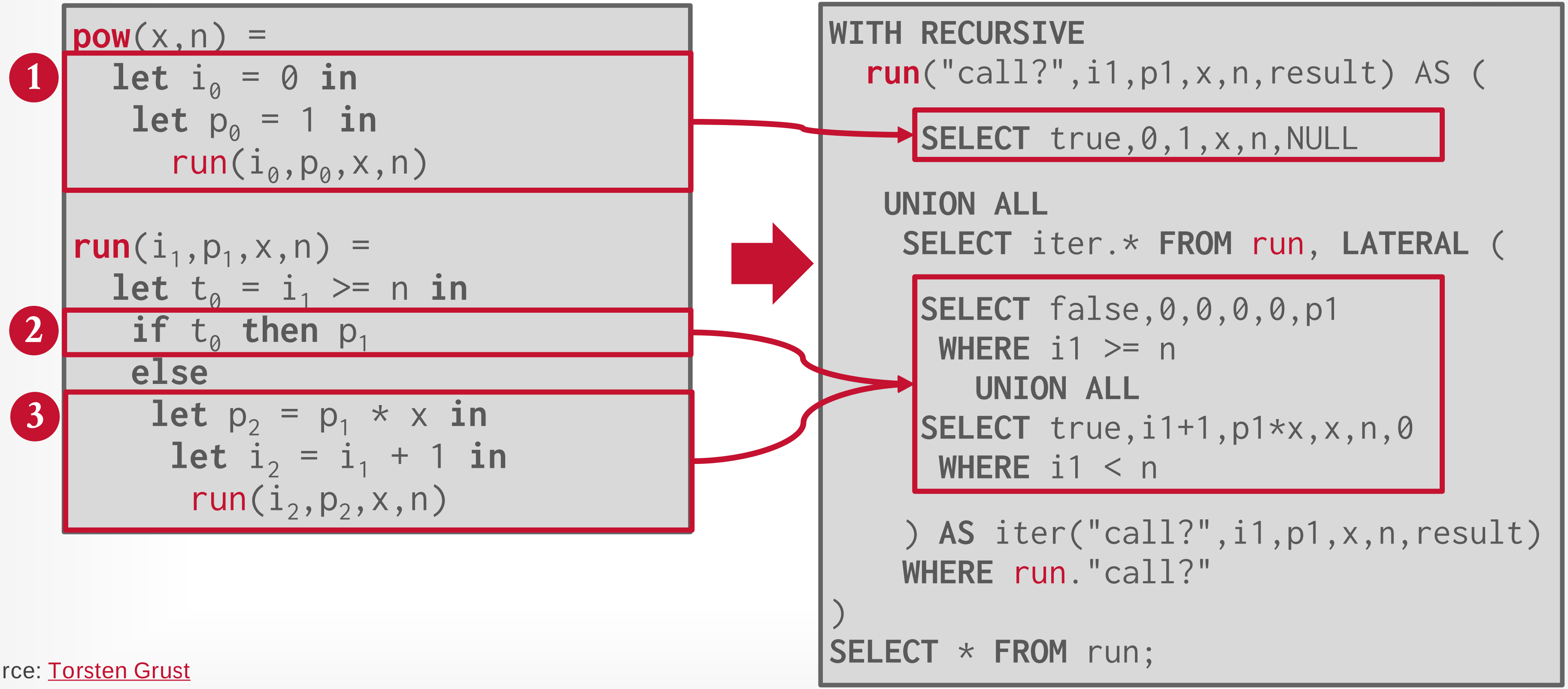
STEP #4: WITH RECURSIVE

```
pow(x,n) =
①
  let $i_0$ = 0 in
   let $p_0$ = 1 in
     run($i_0$,$p_0$,x,n)

run($i_1$,$p_1$,x,n) =
  let $t_0$ = $i_1$ >= n in
   if $t_0$ then $p_1$
   else
    let $p_2$ = $p_1$ * 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

STEP #4: WITH RECURSIVE



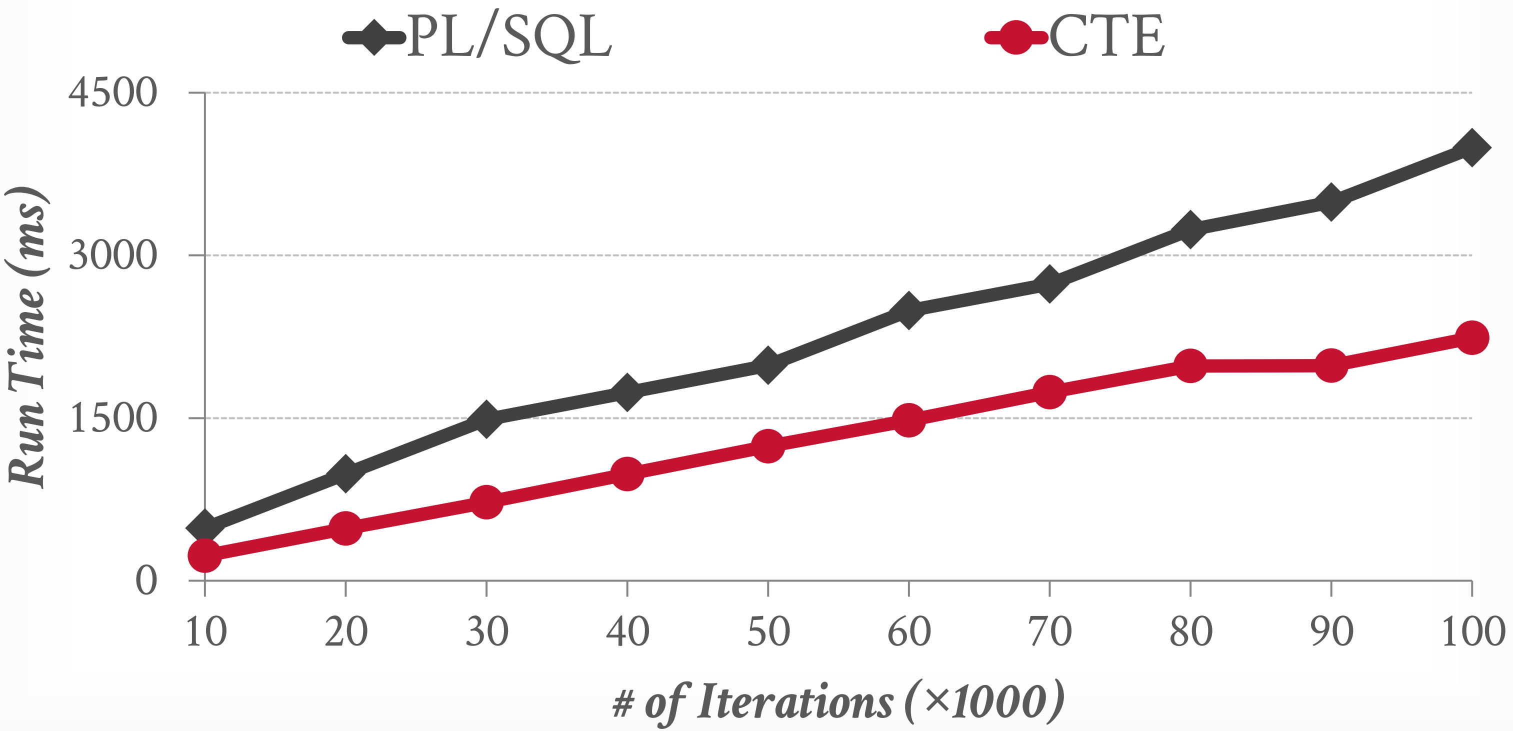
```
pow(x,n) =
  let i_0 = 0 in
   let p_0 = 1 in
     run(i_0,p_0,x,n)

run(i_1,p_1,x,n) =
  let t_0 = i_1 >= n in
   if t_0 then p_1
   else
    let p_2 = p_1 * 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.

DEAR USER-DEFINED FUNCTIONS, INLINING ISN'T WORKING OUT SO GREAT FOR US. LET'S TRY BATCHING TO MAKE OUR RELATIONSHIP WORK. SINCERELY, SQL
CIDR 2024

UDF BATCHING

```
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;
```

UDF 20b from ProcBench

Source: Kai Franz

UDF BATCHING

```
CREATE FUNCTION getManufact(item_id INT)
 RETURNS CHAR(50) AS $$
 DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
 BEGIN
  man := '';
  cnt1 Q1 (SELECT COUNT(*)
             FROM store_sales_history, date_dim
            WHERE ss_item_sk = item_id
              AND d_date_sk = ss_sold_date_sk
              AND d_year = 2023);
  cnt2 Q2 (SELECT COUNT(*)
             FROM catalog_sales_history, date_dim
            WHERE cs_item_sk = item_id
              AND d_date_sk = cs_sold_date_sk
              AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0)
   THEN man Q3 (SELECT i_manufact FROM item
                 WHERE i_item_sk = item_id);
   ELSE man := 'outdated item';
  END IF;
  RETURN man;
 END $$ LANGUAGE PLPGSQL;
```

UDF 20b from ProcBench

Source: Kai Franz

UDF BATCHING

```
CREATE FUNCTION getManufact(item_id INT)
 RETURNS CHAR(50) AS $$
 DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
 BEGIN
  man := '';
  cnt1 Q1 (SELECT COUNT(*)
             FROM store_sales_history, date_dim
            WHERE ss_item_sk = item_id
              AND d_date_sk = ss_sold_date_sk
              AND d_year = 2023);
  cnt2 Q2 (SELECT COUNT(*)
             FROM catalog_sales_history, date_dim
            WHERE cs_item_sk = item_id
              AND d_date_sk = cs_sold_date_sk
              AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0)
   THEN man Q3 (SELECT i_manufact FROM item
                 WHERE i_item_sk = item_id);
   ELSE man := 'outdated item';
  END IF;
  RETURN ma
 END $$ LAN
```

```
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';
```

UDF 20b from ProcBench

Source: Kai Franz

UDF BATCHING

```
CREATE FUNCTION getManufact(item_id INT)
 RETURNS CHAR(50) AS $$
 DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
 BEGIN
  man := '';
  cnt1 Q1 (SELECT COUNT(*)
             FROM store_sales_history, date_dim
            WHERE ss_item_sk = item_id
              AND d_date_sk = ss_sold_date_sk
              AND d_year = 2023);
  cnt2 Q2 (SELECT COUNT(*)
             FROM catalog_sales_history, date_dim
            WHERE cs_item_sk = item_id
              AND d_date_sk = cs_sold_date_sk
              AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0)
   THEN man Q3 (SELECT i_manufact FROM item
                  WHERE i_item_sk = item_id);
   ELSE man := 'outdated item';
  END IF;
  RETURN ma
 END $$ LAN
```

```
SELECT ws_item_sk
  F Q4 (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';
```

UDF 20b from ProcBench

Source: Kai Franz

UDF BATCHING

```
CREATE FUNCTION getManufact(item_id INT)
 RETURNS CHAR(50) AS $$
 DECLARE
  man CHAR(50); cnt1 INT; cnt2 INT;
 BEGIN
  man := '';
  cnt1 Q1 (SELECT COUNT(*)
             FROM store_sales_history, date_dim
            WHERE ss_item_sk = item_id
              AND d_date_sk = ss_sold_date_sk
              AND d_year = 2023);
  cnt2 Q2 (SELECT COUNT(*)
             FROM catalog_sales_history, date_dim
            WHERE cs_item_sk = item_id
              AND d_date_sk = cs_sold_date_sk
              AND d_year = 2023);
  IF (cnt1 > 0 AND cnt2 > 0)
   THEN man Q3 (SELECT i_manufact FROM item
                  WHERE i_item_sk = item_id);
   ELSE man := 'outdated item';
  END IF;
  RETURN ma
 END $$ LAN
```

```
SELECT ws_item_sk
  F Q4 (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 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) WHERE NOT returned;
UPDATE state SET cnt2 = (Q2) WHERE NOT returned;
UPDATE state SET p = COALESCE(cnt1>0 AND cnt2>0, FALSE)
 WHERE NOT returned;
UPDATE state SET man = (Q3) 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';
```

Source: Kai Franz

PROCEDURAL EXTENSIONS OF SQL

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

PROCEDURAL EXTENSIONS OF SQL:
UNDERSTANDING THEIR USAGE IN THE WILD
VLDB 2021

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

		UDFs												
	Method	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.

DEAR USER-DEFINED FUNCTIONS, INLINING ISN'T WORKING OUT SO GREAT FOR US. LET'S TRY BATCHING TO MAKE OUR RELATIONSHIP WORK. SINCERELY, SQL
CIDR 2024

UDF BATCHING VS. INLINING

		UDFs												
	Method	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	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

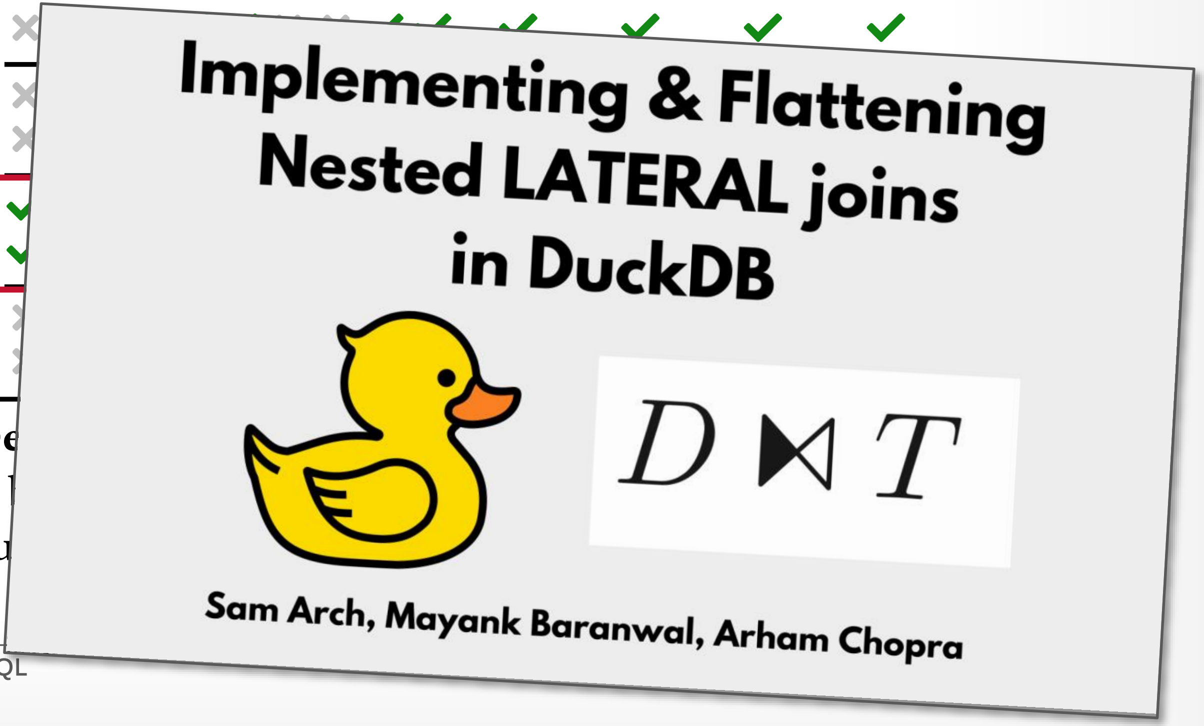
Table 1: Subquery Decorrelation – Whether a given UDF's subqueries could be decorrelated by a DBMS after inlining or batching. Symbol (✓) indicates that some, but not all subqueries could be decorrelated.

DEAR USER-DEFINED FUNCTIONS, INLINING ISN'T WORKING OUT SO GREAT FOR US. LET'S TRY BATCHING TO MAKE OUR RELATIONSHIP WORK. SINCERELY, SQL
CIDR 2024

UDF BATCHING VS. INLINING

	Method	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 De
could be decorrelated
indicates that some, bu



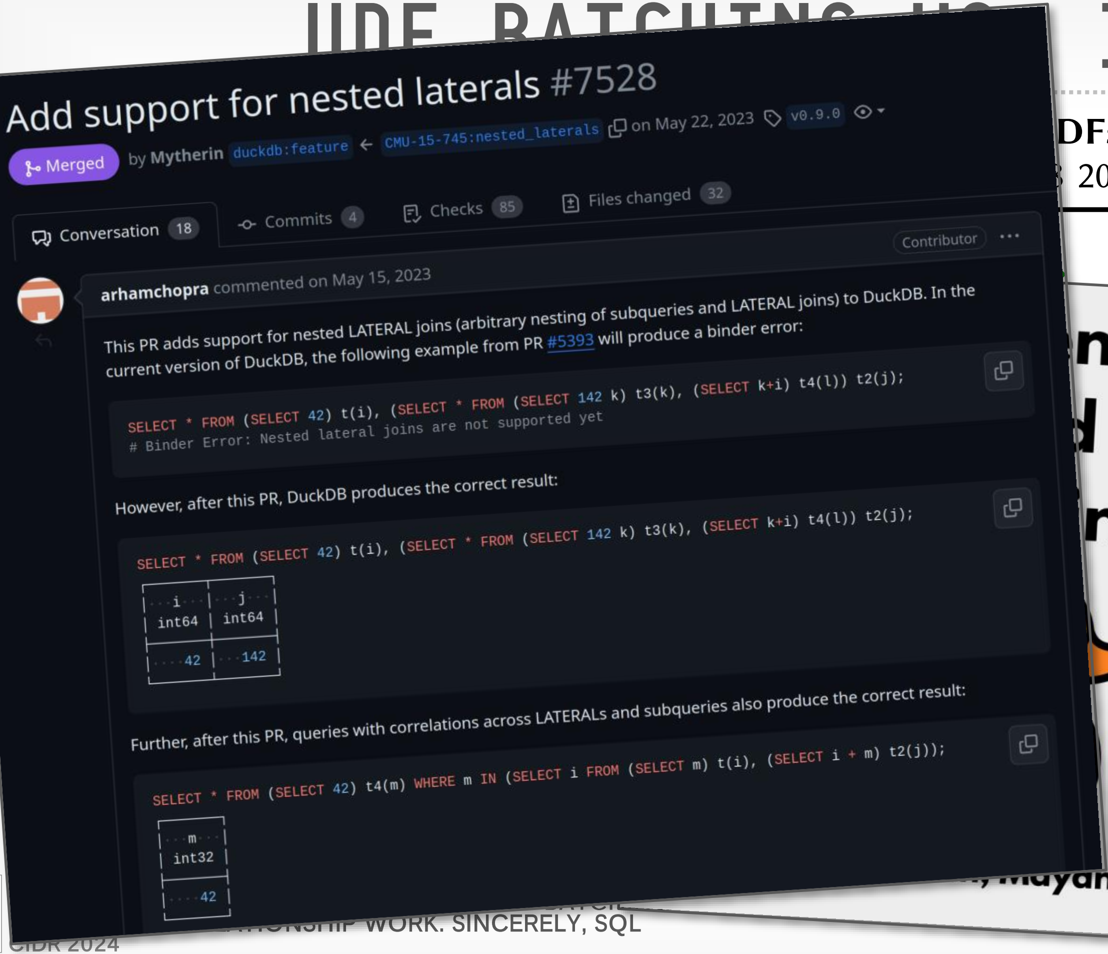
Implementing & Flattening Nested LATERAL joins in DuckDB

$D \bowtie T$

Sam Arch, Mayank Baranwal, Arham Chopra

DEAR USER-DEFINED FUNCTIONS, INLINING ISN'T WORKING OUT SO GREAT FOR US. LET'S TRY BATCH
TO MAKE OUR RELATIONSHIP WORK. SINCERELY, SQL
CIDR 2024

UDF BATCHING VS. INLINING



Add support for nested laterals #7528

Merged by **Mytherin** `duckdb:feature` ← `CMU-15-745:nested_laterals` on May 22, 2023 v0.9.0

arhamchopra commented on May 15, 2023

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 #5393 will produce a binder error:

```
SELECT * FROM (SELECT 42) t(i), (SELECT * FROM (SELECT 142 k) t3(k), (SELECT k+i) t4(l)) t2(j);
# Binder Error: Nested lateral joins are not supported yet
```

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

```
SELECT * FROM (SELECT 42) t(i), (SELECT * FROM (SELECT 142 k) t3(k), (SELECT k+i) t4(l)) t2(j);
┌───────┬───────┐
│   i   │   j   │
│ int64 │ int64 │
├───────┼───────┤
│    42 │   142 │
└───────┴───────┘
```

Further, after this PR, queries with correlations across LATERALs and subqueries also produce the correct result:

```
SELECT * FROM (SELECT 42) t4(m) WHERE m IN (SELECT i FROM (SELECT m) t(i), (SELECT i + m) t2(j));
┌───────┐
│   m   │
│ int32 │
├───────┤
│    42 │
```

DFs

	20a_q1	20a_q2	20b_q1	20b_q2
	✓	✓	✗	✗
	✓	✓	✓	✓

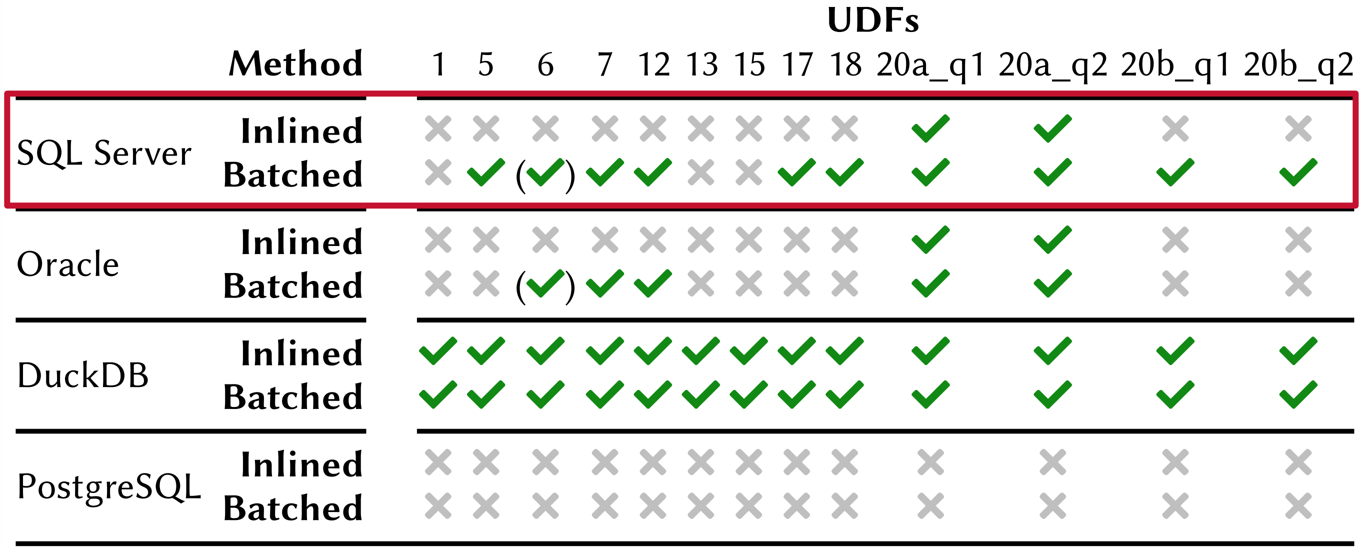
nting & Flattening
d LATERAL joins
n DuckDB

ayank Baranwal, Arham Chopra

CIDR 2024

ATIONSHIP WORK. SINCERELY, SQL

UDF BATCHING VS. INLINING



		UDFs												
	Method	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.

DEAR USER-DEFINED FUNCTIONS, INLINING ISN'T WORKING OUT SO GREAT FOR US. LET'S TRY BATCHING TO MAKE OUR RELATIONSHIP WORK. SINCERELY, SQL
CIDR 2024

DECORRELATION OF SUBQUERIES (MSSQL)

Algebraic rewrite rules for APPLY

$$R\ \mathcal{A}^{\otimes}\ E \;=\; R \otimes_{\text{true}} E, \tag{1}$$

if no parameters in E resolved from R

$$R\ \mathcal{A}^{\otimes}\ (\sigma_p E) \;=\; R \otimes_p E, \tag{2}$$

if no parameters in E resolved from R

$$R\,\mathcal{A}^{\times}\ (\sigma_p E) \;=\; \sigma_p(R\,\mathcal{A}^{\times}\ E) \tag{3}$$

$$R\,\mathcal{A}^{\times}\ (\pi_v E) \;=\; \pi_{v \,\cup\, \text{columns}(R)}(R\,\mathcal{A}^{\times}\ E) \tag{4}$$

$$R\,\mathcal{A}^{\times}\ (E_1 \cup E_2) \;=\; (R\,\mathcal{A}^{\times}\ E_1) \cup (R\,\mathcal{A}^{\times}\ E_2) \tag{5}$$

$$R\,\mathcal{A}^{\times}\ (E_1 - E_2) \;=\; (R\,\mathcal{A}^{\times}\ E_1) - (R\,\mathcal{A}^{\times}\ E_2) \tag{6}$$

$$R\,\mathcal{A}^{\times}\ (E_1 \times E_2) \;=\; (R\,\mathcal{A}^{\times}\ E_1) \bowtie_{R.key} (R\,\mathcal{A}^{\times}\ E_2) \tag{7}$$

$$R\,\mathcal{A}^{\times}\ (\mathcal{G}_{A,F} E) \;=\; \mathcal{G}_{A \,\cup\, \text{columns}(R),F}(R\,\mathcal{A}^{\times}\ E) \tag{8}$$

$$R\,\mathcal{A}^{\times}\ (\mathcal{G}^1_F E) \;=\; \mathcal{G}_{\text{columns}(R),F'}(R\,\mathcal{A}^{\text{LOJ}}\ E) \tag{9}$$

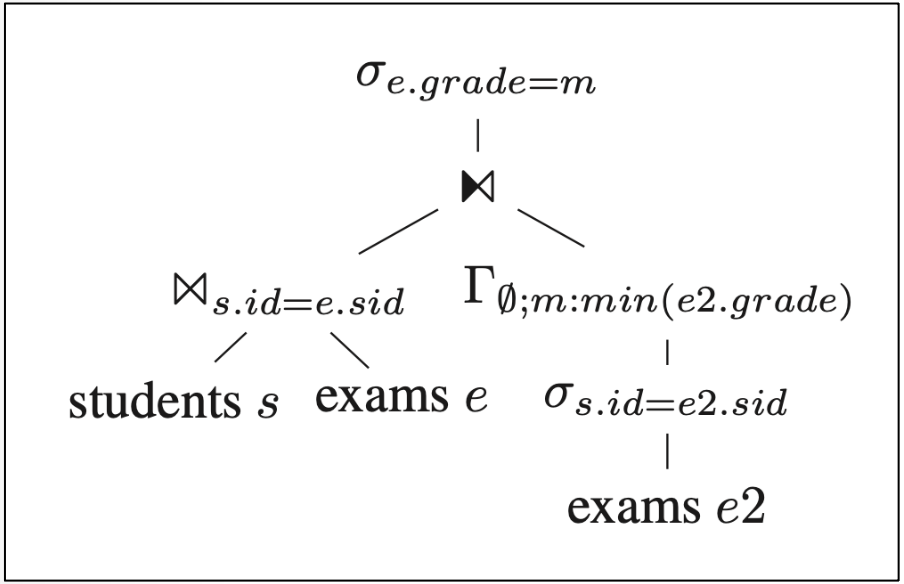
Execute the rewrite rules where applicable

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

ORTHOGONAL OPTIMIZATION OF SUBQUERIES AND AGGREGATION
SIGMOD 2001

DECORRELATION OF SUBQUERIES (GERMANS)

Dependent Join Operator



Introduces a new "Dependent Join" operator into the Query Plan DAG

Systematically decorrelates any subquery

UNNESTING ARBITRARY QUERIES
BTW 2015

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.

NEXT CLASS

Database Networking Protocols

And a little bit about kernel bypass methods…