TODAY’S AGENDA

Background
UDF In-lining
UDF CTE Conversion
Until now, we have assumed that all of the logic for an application is located in the application itself.

The application has a "conversation" with the DBMS to store/retrieve data.
→ Protocols: JDBC, ODBC
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL
Program Logic
SQL
Program Logic :

COMMIT
CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
::
COMMIT
```

Parser
Planner
Optimizer
Query Execution
CONVERSATIONAL DATABASE API

Application

BEGIN

$SQL$
Program Logic

$SQL$
Program Logic

::

COMMIT

Parser

Planner

Optimizer

Query Execution
CONVERSATIONAL DATABASE API

**Application**

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  :
  COMMIT
```
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL

Program Logic

SQL

Program Logic

::

COMMIT

Parser

Planner

Optimizer

Query Execution
EMBEDDED DATABASE LOGIC

Move application logic into the DBMS to avoid multiple network round-trips and to extend the functionality of the DBMS.

Potential Benefits
→ Efficiency
→ Reuse
Embedded Database Logic

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

:

COMMIT
EMBEDDED DATABASE LOGIC

Application

CALL PROC(x=99)

PROC(x)

BEGIN
  SQL
  Program Logic
  SQL
  Program Logic :
  COMMIT
EMBEDDED DATABASE LOGIC

User-Defined Functions (UDFs)
Stored Procedures
Triggers
User-Defined Types (UDTs)
User-Defined Aggregates (UDAs)
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)
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 UDFs as black boxes. → Unable to estimate cost if you don't know what a UDF is going to do when you run it.

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 PERFORMANCE

Microsoft SQL Server

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

```
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
```

Source: Karthik Ramachandra
TPC-H Q12 using a UDF (SF=1).

→ Original Query: 0.8 sec

→ Query + UDF: 13 hr 30 min

Source: Karthik Ramachandra
MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.
2008 – People realize that UDFs are "evil".

Source: Karthik Ramachandra
**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(replicate(@char, @len) + @val, @len)
end
```

**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 2.14 ms for the system calls and 3875 ms for the UDF. That's a 19x increase.

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

```sql
select max(dbo.PadLeft(c.name, 100, '0'))
from msdb.sys.columns c
```

Source: [Karthik Ramachandra](https://www.karthikramachandra.com)

---

15-721 (Spring 2020)
MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.
2008 – People realize that UDFs are "evil".
2010 – Microsoft acknowledges that UDFs are evil.

Source: Karthik Ramachandra
MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.

2008 – People realize that UDFs are “evil”.

2010 – Microsoft acknowledges that UDFs are evil.

Source: Karthik Ramachandra
MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.
2008 – People realize that UDFs are "evil".
2010 – Microsoft acknowledges that UDFs are evil.
2014 – UDF decorrelation research @ IIT-B.
2015 – Froid project begins @ MSFT Gray Lab.
2018 – Froid added to SQL Server 2019.

Source: Karthik Ramachandra
MICROSOFT SQL SERVER UDF HISTORY

- 2001 – Microsoft adds TSQL Scalar UDFs.
- 2008 – People realize that UDFs are “evil”.
- 2010 – Microsoft acknowledges that UDFs are evil.
- 2014 – UDF decorrelation research @ IIT.
- 2015 – Froid project begins @ MSFT Gray Lab.

Source: Karthik Ramachandra
Automatically convert UDFs into relational 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.
SUB-QUERIES – REWRITE

SELECT name FROM sailors AS S
WHERE EXISTS ( 
    SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
    AND R.day = '2020-04-22'
)

SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2020-04-22'
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.
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 Through Query Optimizer
STEP #1 – TRANSFORM STATEMENTS

**Imperative Statements**

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

**SQL Statements**

- `SELECT 'Platinum' 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
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
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

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 #4 – INLINE EXPRESSION

Original Query

```sql
SELECT c_custkey, 
cust_level(c_custkey)
FROM customer
```
STEP #4 – INLINE EXPRESSION

Original Query

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

Modified 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 #4 – INLINE EXPRESSION**

**Original Query**

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

```sql
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

```sql
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_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

SELECT `returnVal` FROM (
  SELECT CASE WHEN @x > 1000
    THEN 'high'
    ELSE 'low' END AS `val`
  AS `DT1`
) AS `DT1`
OUTER APPLY (
  SELECT `DT1`.val + ' value'
  AS `returnVal`
) AS `DT2`
**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`

**Dynamic Slicing**

`SELECT returnVal FROM (SELECT CASE WHEN @x > 1000 THEN 'high' ELSE 'low' END AS val) AS DT1 OUTER APPLY (SELECT DT1.val + ' value' AS returnVal) AS DT2`

**Const Propagation & Folding**

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

**Dead Code Elimination**

`BEGIN RETURN 'high value'; END`

**Froid**
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.
### APPLICABILITY / COVERAGE

<table>
<thead>
<tr>
<th>Workload</th>
<th># of Scalar UDFs</th>
<th>Froid Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload 1</td>
<td>178</td>
<td>150</td>
</tr>
<tr>
<td>Workload 2</td>
<td>90</td>
<td>82</td>
</tr>
<tr>
<td>Workload 3</td>
<td>22</td>
<td>21</td>
</tr>
</tbody>
</table>
UDF IMPROVEMENT STUDY

Table: 100k Tuples

Improvement Factor

Source: Karthik Ramachandra

15-721 (Spring 2020)
Rewrite UDFs into plain SQL commands.

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

DBMS Agnostic

→ Can be implemented as a rewrite middleware layer on top of any DBMS that supports CTEs.
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 \textbf{WITH RECURSIVE}
Step #5 – Run Through Query Optimizer
STEP #1 – STATIC SINGLE ASSIGNMENT

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

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

Source: Torsten Grust
STEP #2 – ADMINISTRATIVE NORMAL FORM

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

\[\text{body}(i_1, p_1, x, n) = \]
\[
\begin{align*}
&\text{let } p_2 = p_1 \times x \text{ in} \\
&\text{let } i_2 = i_1 + 1 \text{ in} \\
&\text{while}(i_2, p_2, x, n)
\end{align*}
\]

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

\[\text{body}(i_1, p_1, x, n) = \]
\[
\begin{align*}
&\text{let } p_2 = p_1 \times x \text{ in} \\
&\text{let } i_2 = i_1 + 1 \text{ in} \\
&\text{while}(i_2, p_2, x, n)
\end{align*}
\]

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

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

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

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

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

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

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

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

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

\section*{STEP #4 – WITH RECURSIVE}

\begin{verbatim}
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
      ) AS iter("call?",i1,p1,x,n,result)
   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;
\end{verbatim}
STEP #4 – WITH RECURSIVE

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

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

WITH RECURSIVE
\[
\text{run}("call?",i1,p1,x,n,result) \text{ AS (}
\begin{align*}
&\text{SELECT true,0,1,x,n,NULL} \\
&\text{UNION ALL} \\
&\text{SELECT iter.\_star \ FROM run, LATERAL (}
\begin{align*}
&\text{SELECT false,0,0,0,0,p1} \\
&\text{WHERE i1 \geq n} \\
&\text{UNION ALL} \\
&\text{SELECT true,i1+1,p1*x,x,n,0} \\
&\text{WHERE i1 < n}
\end{align*}
\)
\text{ AS iter("call?",i1,p1,x,n,result)} \\
\text{WHERE run."call?"}
\)
\text{SELECT \_star \ FROM run;}
\]

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

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

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

Source: Torsten Grust
UDFs-TO-CTEs EVALUATION

POW UDF on Postgres v11.3

Source: Torsten Grust

15-721 (Spring 2020)
PARTING THOUGHTS

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

Another optimization approach is to compile the UDF into machine code.
→ This does not solve the optimizer's cost model problem.
NEXT CLASS

Last Lecture: Databases on New Hardware