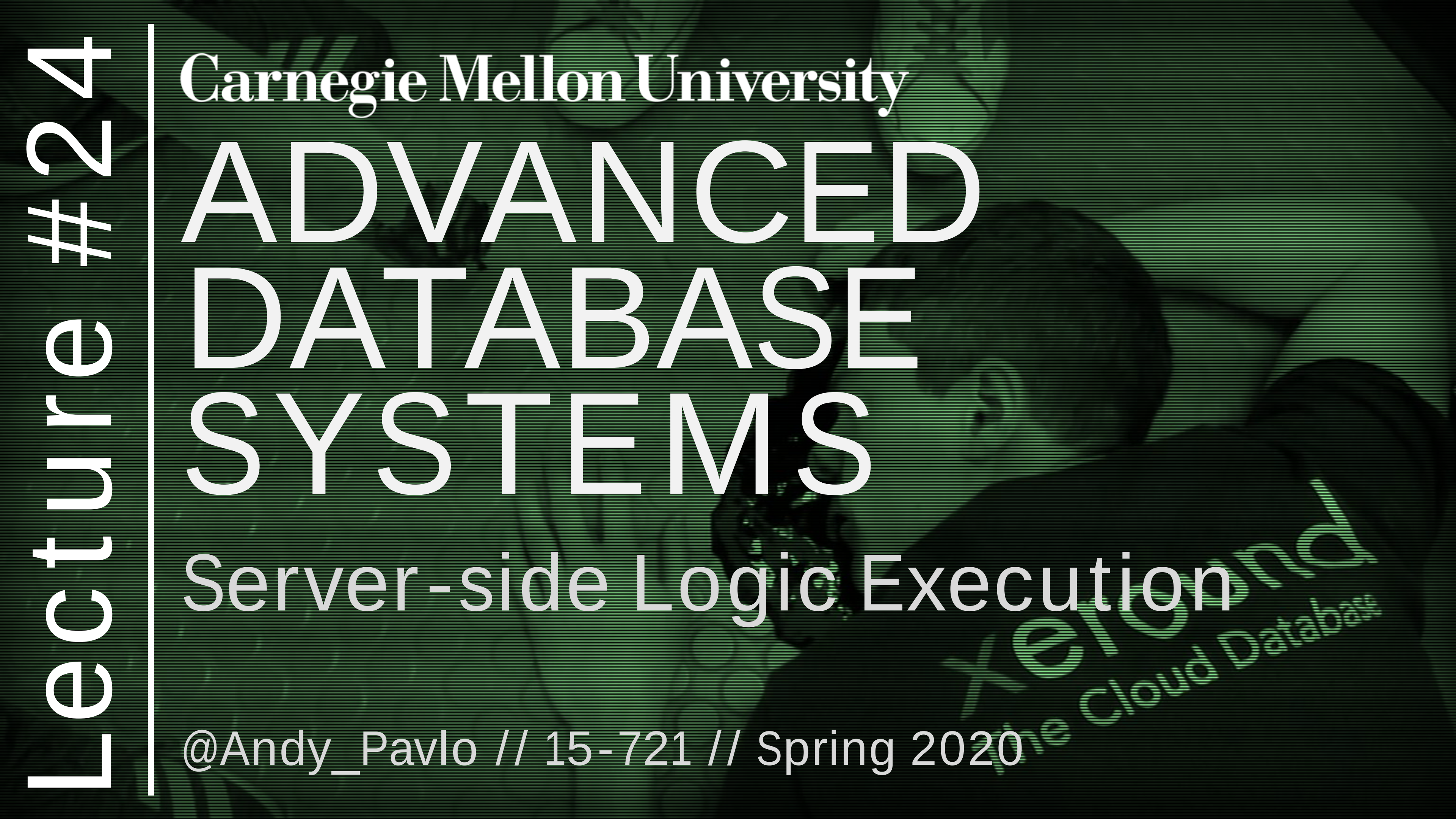


Lecture #24

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Server-side Logic Execution

@Andy_Pavlo // 15-721 // Spring 2020

TODAY'S AGENDA

Background

UDF In-lining

UDF CTE Conversion

OBSERVATION

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.

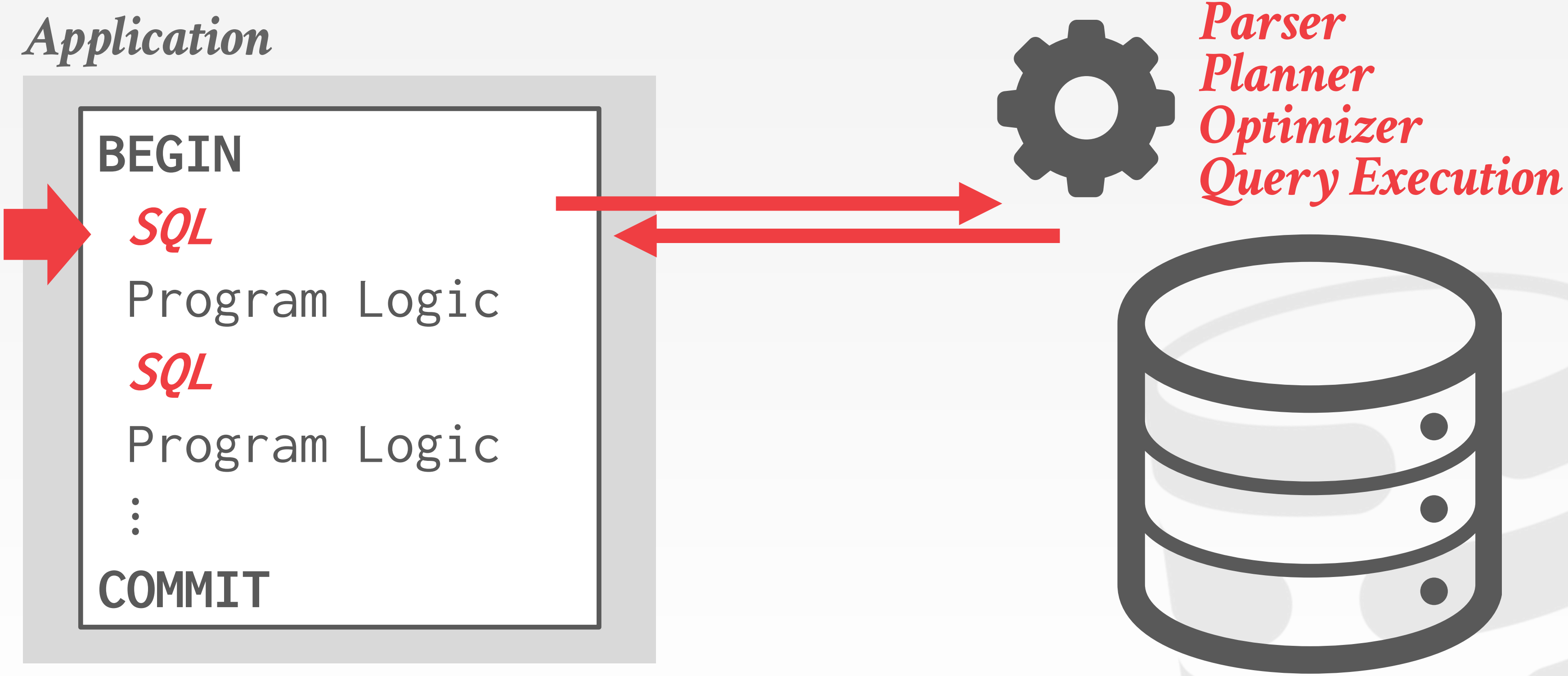
→ Protocols: JDBC, ODBC

CONVERSATIONAL DATABASE API

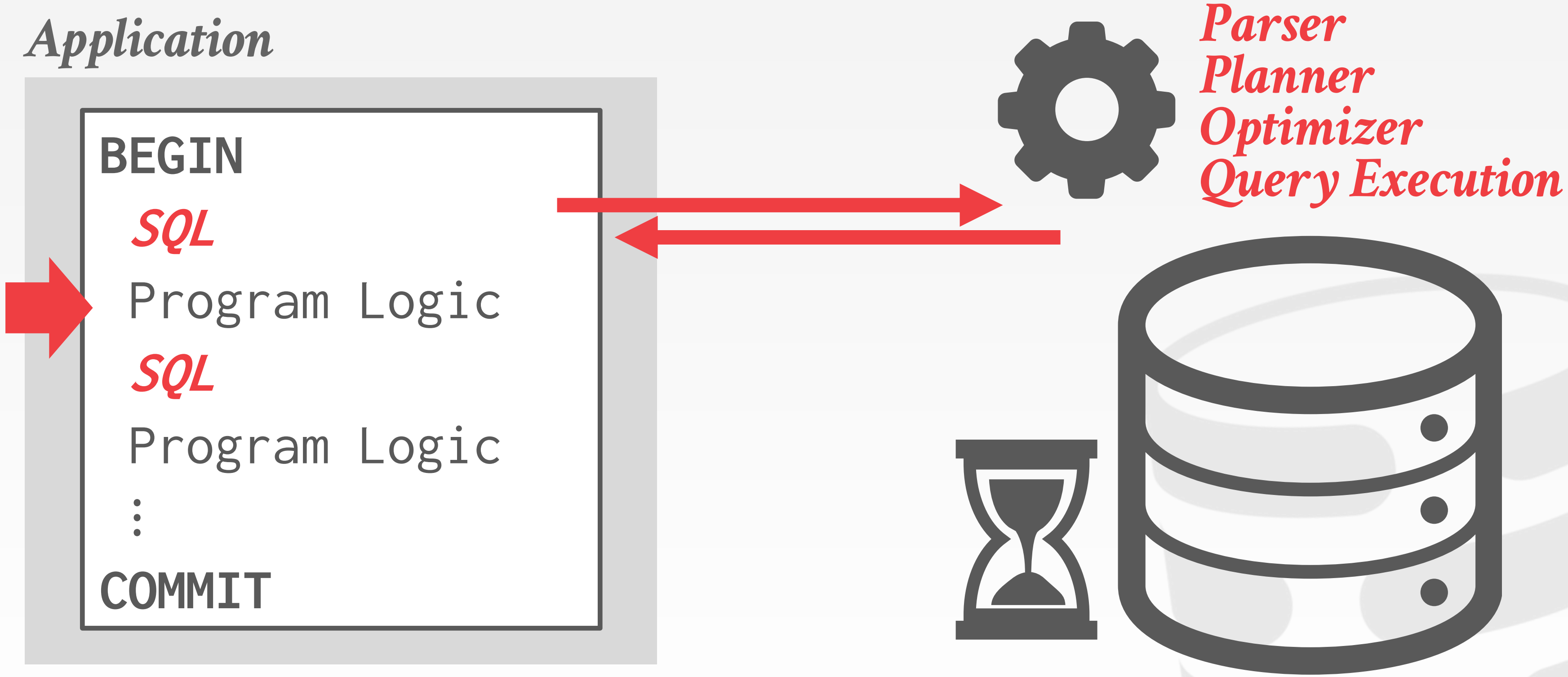
Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

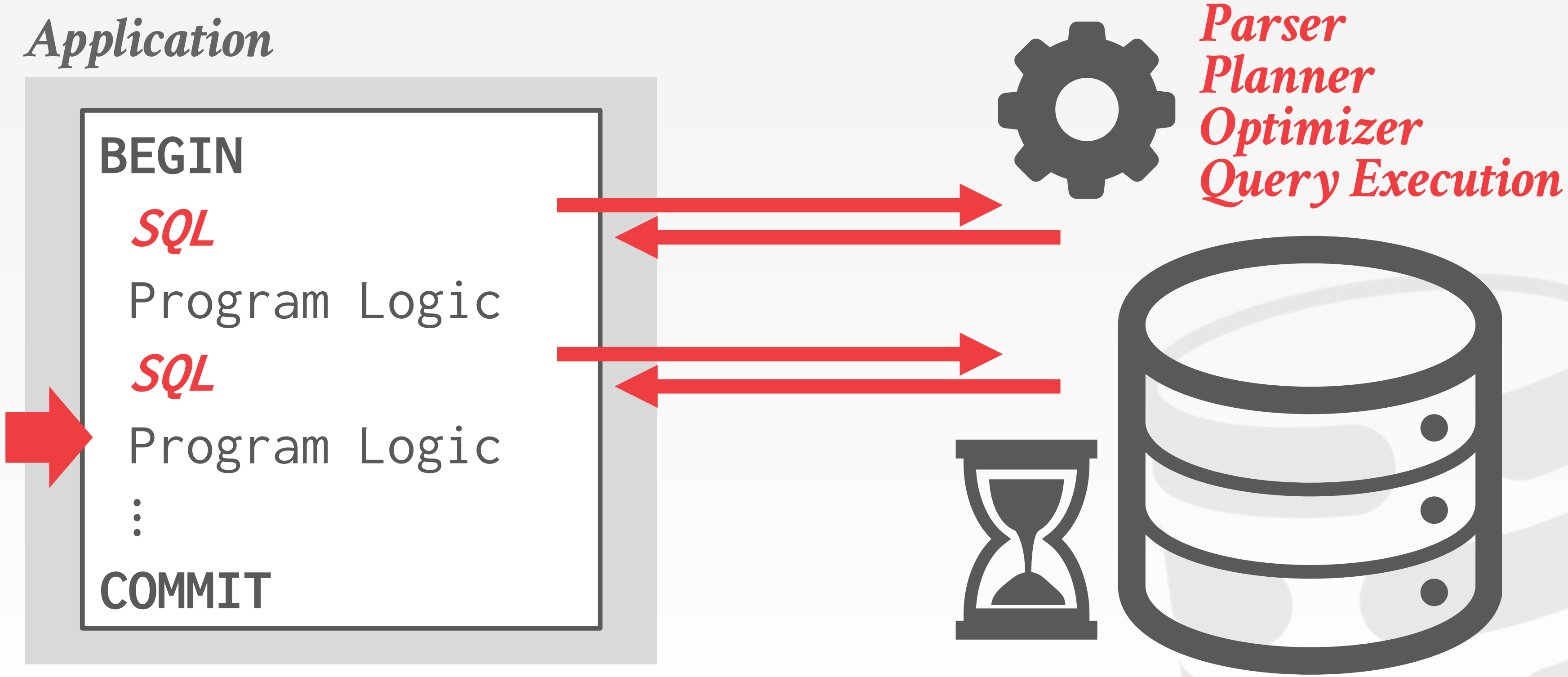
CONVERSATIONAL DATABASE API



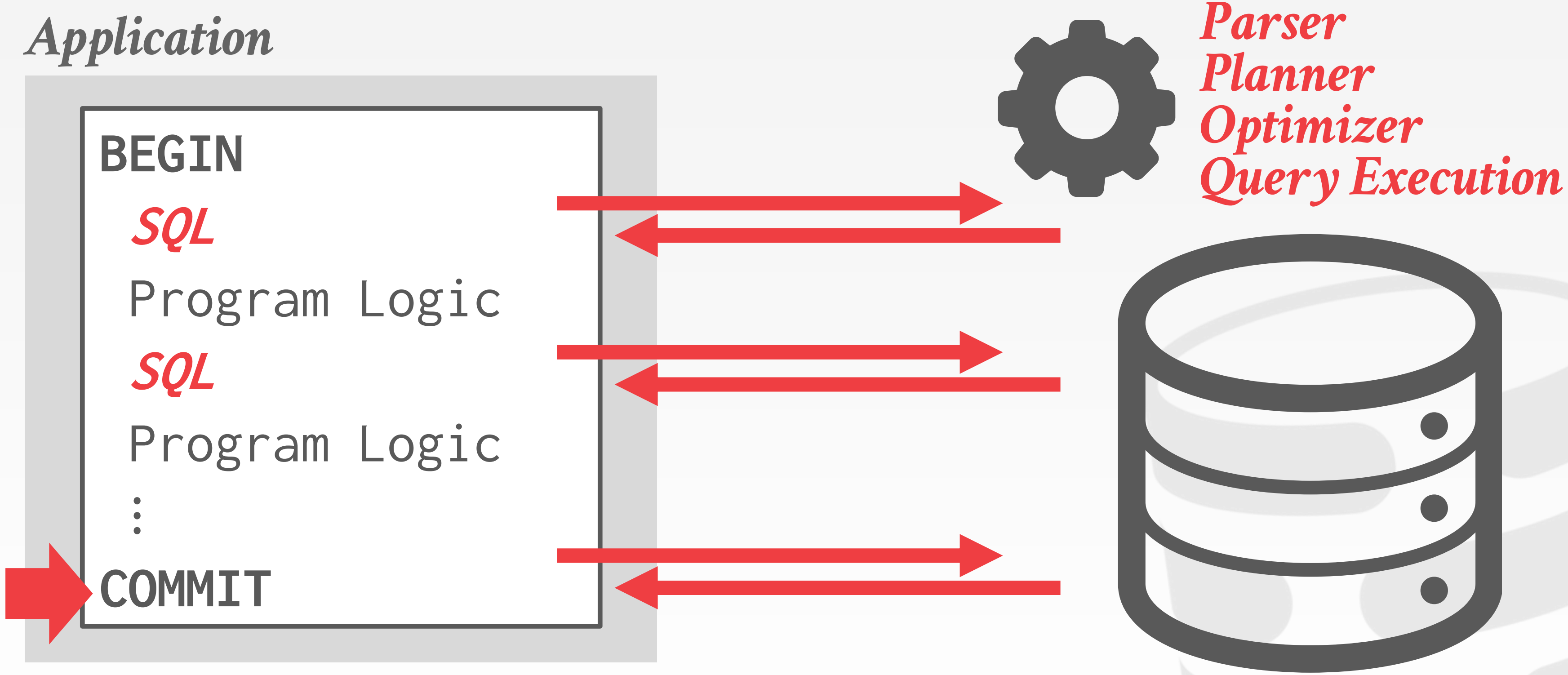
CONVERSATIONAL DATABASE API



CONVERSATIONAL DATABASE API



CONVERSATIONAL DATABASE API



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

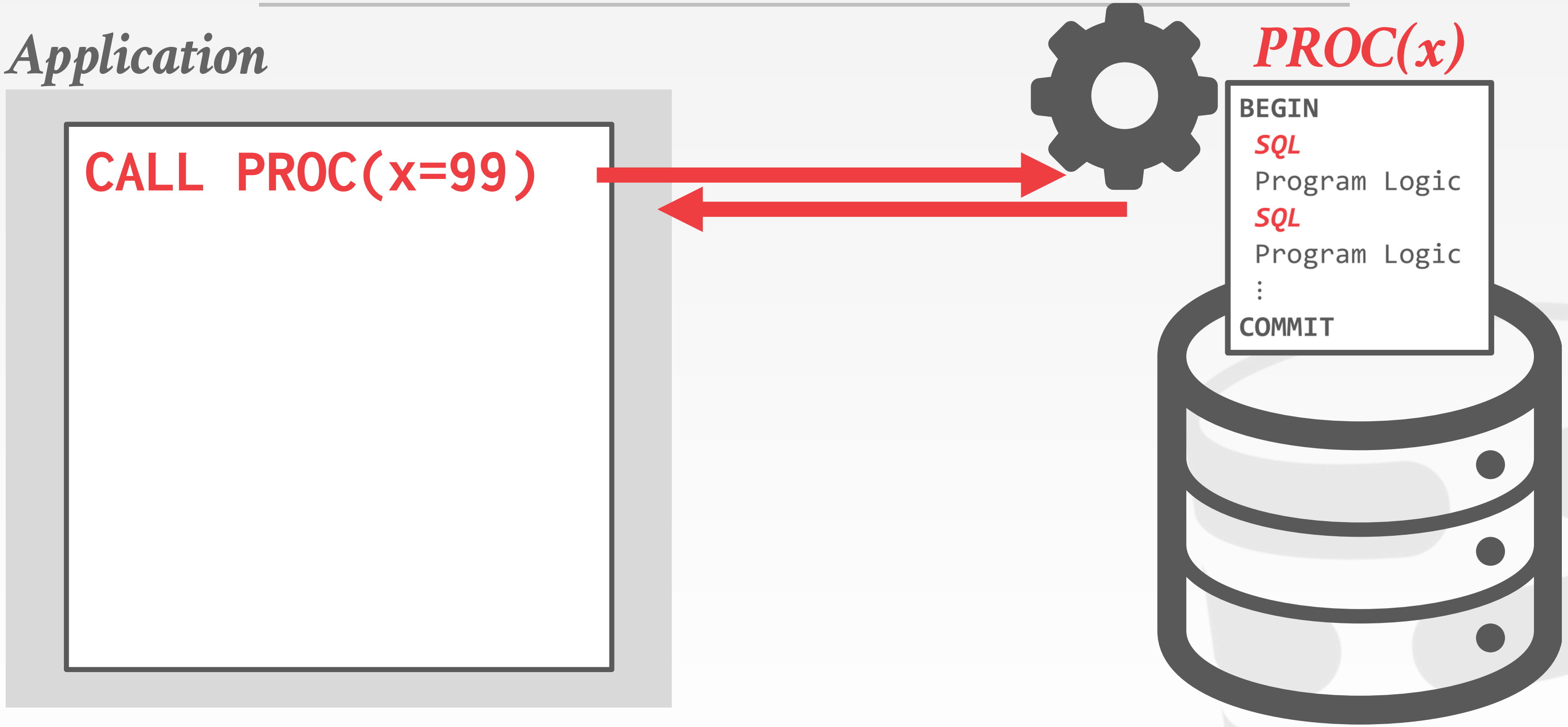
→ Reuse

EMBEDDED DATABASE LOGIC

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

EMBEDDED DATABASE LOGIC



EMBEDDED DATABASE LOGIC

User-Defined Functions (UDFs)

Stored Procedures

Triggers

User-Defined Types (UDTs)

User-Defined Aggregates (UDAs)

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)

UDF EXAMPLE

```
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

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

MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.

2008 – People realize that UDFs are "evil".

Source: Karthik Ramachandra

...F HISTORY

...DFs.

...vil".

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
```

Source:

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

2001 – Micro

2008 – Peopl

2010 – Micro

Microsoft

Sign In

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):

Source: Karthik Ramachandra

MICROSOFT SQL SERVER UDF HISTORY

2001 – Microsoft adds TSQL Scalar UDFs.

2008 – People realize that UDFs are "evil".

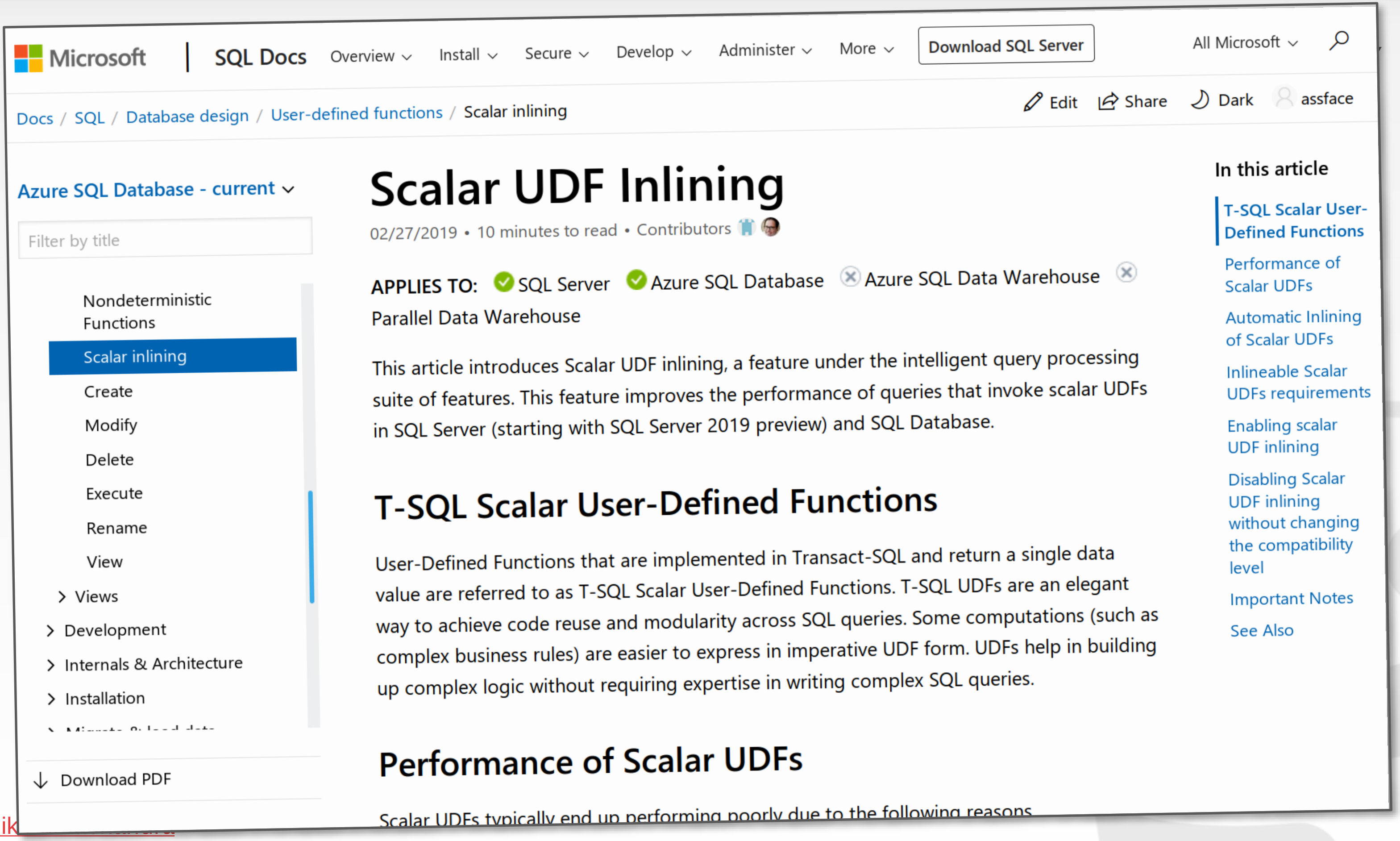
2010 – Microsoft acknowledges that UDFs are evil.

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 Docs Overview Install Secure Develop Administer More Download SQL Server All Microsoft

Docs / SQL / Database design / User-defined functions / Scalar inlining

Edit Share Dark assface

Azure SQL Database - current

Filter by title

- Nondeterministic Functions
- Scalar inlining
- Create
- Modify
- Delete
- Execute
- Rename
- View
- Views
- Development
- Internals & Architecture
- Installation
- Migrate & load data

Download PDF

Scalar UDF Inlining

02/27/2019 • 10 minutes to read • Contributors

APPLIES TO: SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This article introduces Scalar UDF inlining, a feature under the intelligent query processing suite of features. This feature improves the performance of queries that invoke scalar UDFs in SQL Server (starting with SQL Server 2019 preview) and SQL Database.

T-SQL Scalar User-Defined Functions

User-Defined Functions that are implemented in Transact-SQL and return a single data value are referred to as T-SQL Scalar User-Defined Functions. T-SQL UDFs are an elegant way to achieve code reuse and modularity across SQL queries. Some computations (such as complex business rules) are easier to express in imperative UDF form. UDFs help in building up complex logic without requiring expertise in writing complex SQL queries.

Performance of Scalar UDFs

Scalar UDFs typically end up performing poorly due to the following reasons

In this article

- T-SQL Scalar User-Defined Functions
- Performance of Scalar UDFs
- Automatic Inlining of Scalar UDFs
- Inlineable Scalar UDFs requirements
- Enabling scalar UDF inlining
- Disabling Scalar UDF inlining without changing the compatibility level
- Important Notes
- See Also

Source: Karthik

FROID

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.

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.

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

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
```

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
```

1

2

```
(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
  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 2 3

```
(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
  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 2 3 4

```
(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 #4 – INLINE EXPRESSION

Original Query

```
SELECT c_custkey,
       cust_level(c_custkey)
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 #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;
```

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
```

Froid

```
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) 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
```

Froid

```
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) DT2
```

Dynamic Slicing

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

```
SELECT returnVal FROM
 (SELECT 'high' 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 'high value';
END
```

```
SELECT returnVal FROM
 (SELECT 'high value'
         AS returnVal)
 AS DT1
```

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)

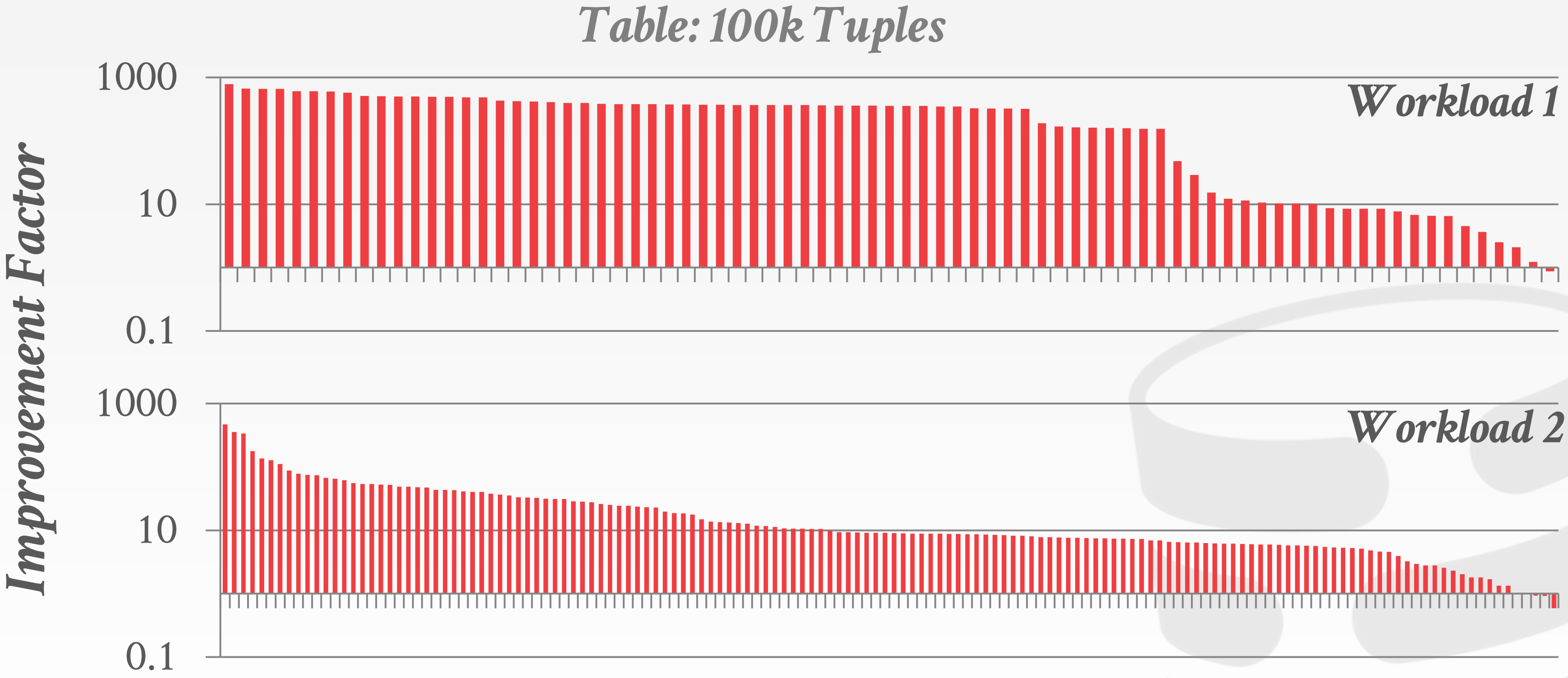
UDF invocation (nested/recursive with configurable depth)

All SQL datatypes.

APPLICABILITY / COVERAGE

	# of Scalar UDFs	Froid Compatible	
Workload 1	178	150	84%
Workload 2	90	82	91%
Workload 3	22	21	95%

UDF IMPROVEMENT STUDY



Source: Karthik Ramachandra

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.

DBMS Agnostic

→ Can be implemented as a rewrite middleware layer on top of any DBMS that supports CTEs.

COMPILING PL/SQL AWAY
CIDR 2020

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):
          i₀ ← 0;
          p₀ ← 0;
 while:   i₁ ← Φ(i₀,i₂);
          p₁ ← Φ(p₀,p₂);
          if i₁ < n then
            goto loop;
          else
            goto exit;
  loop:   p₂ ← p₁ * x;
          i₂ ← i₁ + 1;
          goto while;
  exit:   return p₁;
```

Source: Torsten Grust

STEP #2 – ADMINISTRATIVE NORMAL FORM

```
pow(x,n):
        i0 ← 0;
        p0 ← 0;
 while: i1 ← Φ(i0,i2);
        p1 ← Φ(p0,p2);
        if i1 < n then
          goto loop;
        else
          goto exit;
  loop: p2 ← p1 * x;
        i2 ← i1 + 1;
        goto while;
  exit: return p1;
```

```
pow(x,n) =
  let i0 = 0 in
   let p0 = 1 in
     while(i0,p0,x,n)

while(i1,p1,x,n) =
  let t0 = i1 >= n in
   if t0 then p1
   else body(i1,p1,x,n)

body(i1,p1,x,n) =
  let p2 = p1 * x in
   let i2 = i1 + 1 in
    while(i2,p2,x,n)
```

Source: Torsten Grust

STEP #3 – MUTUAL TO DIRECT RECURSION

```
pow(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
     while(i₀,p₀,x,n)

while(i₁,p₁,x,n) =
  let t₀ = i₁ >= n in
   if t₀ then p₁
   else body(i₁,p₁,x,n)

body(i₁,p₁,x,n) =
  let p₂ = p₁ * x in
   let i₂ = i₁ + 1 in
    while(i₂,p₂,x,n)
```

→

```
pow(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
     run(i₀,p₀,x,n)

run(i₁,p₁,x,n) =
  let t₀ = i₁ >= n in
   if t₀ then p₁
   else
    let p₂ = p₁ * x in
     let i₂ = i₁ + 1 in
      run(i₂,p₂,x,n)
```

Source: Torsten Grust

STEP #4 – WITH RECURSIVE

```
pow(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
     run(i₀,p₀,x,n)

run(i₁,p₁,x,n) =
  let t₀ = i₁ >= n in
   if t₀ then p₁
   else
    let p₂ = p₁ * x in
     let i₂ = i₁ + 1 in
      run(i₂,p₂,x,n)
```

```sql
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 i₀ = 0 in
   let p₀ = 1 in
     run(i₀,p₀,x,n)

run(i₁,p₁,x,n) =
  let t₀ = i₁ >= n in
   if t₀ then p₁
   else
    let p₂ = p₁ * x in
     let i₂ = i₁ + 1 in
      run(i₂,p₂,x,n)
```

1

```
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 i0 = 0 in
    let p0 = 1 in
      run(i0,p0,x,n)

run(i1,p1,x,n) =
  let t0 = i1 >= n in
2   if t0 then p1
    else
3    let p2 = p1 * x in
      let i2 = i1 + 1 in
       run(i2,p2,x,n)
```

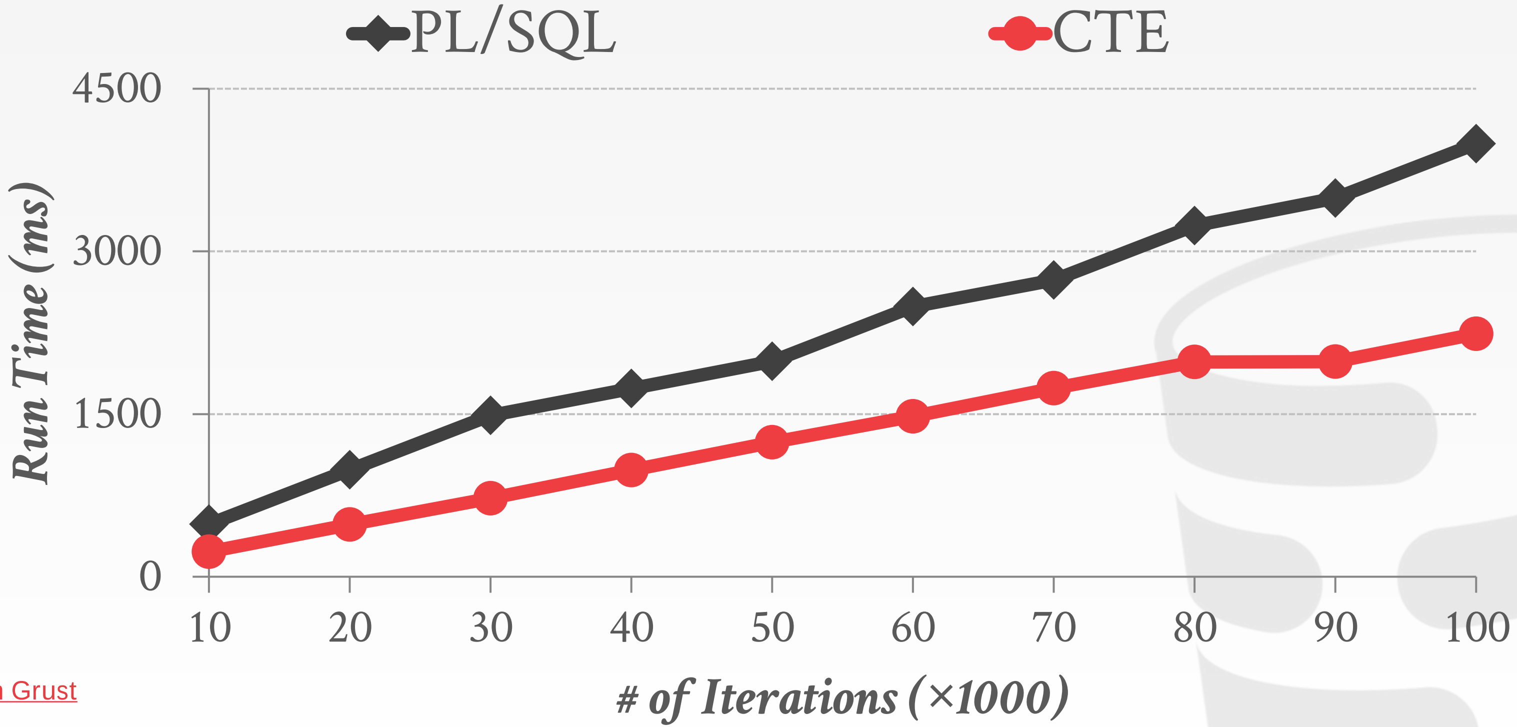
```
WITH RECURSIVE
  run("call?",i1,p1,x,n,result) AS (
    SELECT true,0,1,x,n,NULL

   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

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