

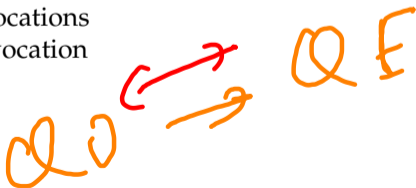
Lecture 24: Server-side Logic Execution



Recap

Adaptive Query Optimization

- The "plan-first execute-second" approach to query planning is notoriously error prone.
- Optimizers should work with the execution engine to provide alternative plan strategies and receive feedback.
- Adaptive techniques now appear in many of the major commercial DBMSs
 - ▶ DB2, Oracle, MSSQL, TeraData
- Approaches
 - ▶ Approach 1: Modify Future Invocations
 - ▶ Approach 2: Replan Current Invocation
 - ▶ Approach 3: Plan Pivot Points



Cost Models

- Using number of tuples processed is a reasonable cost model for in-memory DBMSs.
 - ▶ But computing this is non-trivial.
 - ▶ A combination of sampling + sketches allows the DBMS to achieve accurate estimations.

- relational
 - group by
 H
 K V | P
 V. J

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

client ↔ server

Today's Agenda

- Background
- UDF In-lining
- UDF to CTE Conversion

Background

Conversational Database API

Client

Application

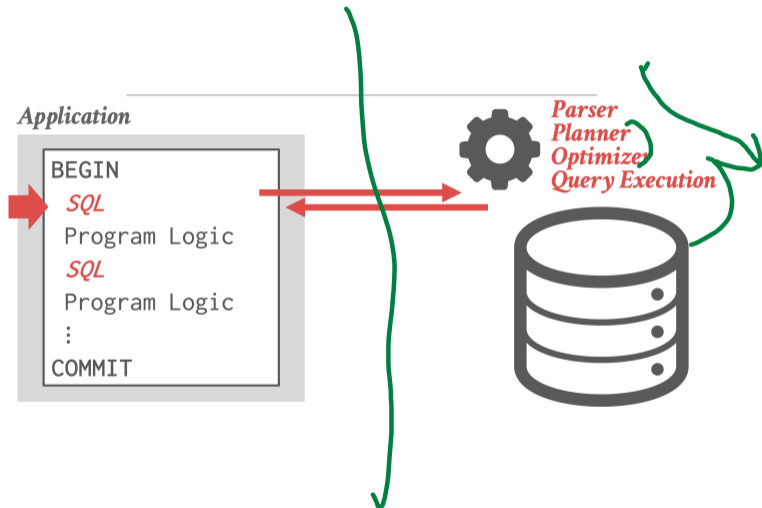
```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  :
  COMMIT
```

Server

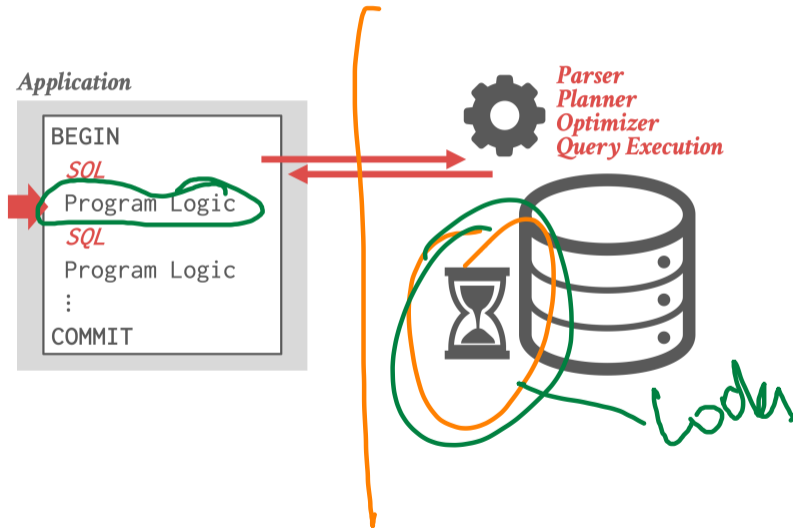


ROP

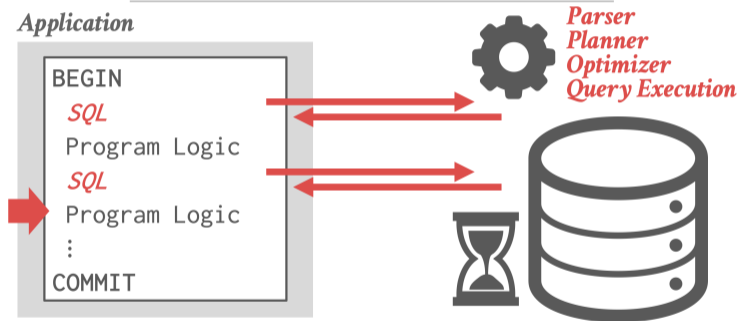
Conversational Database API



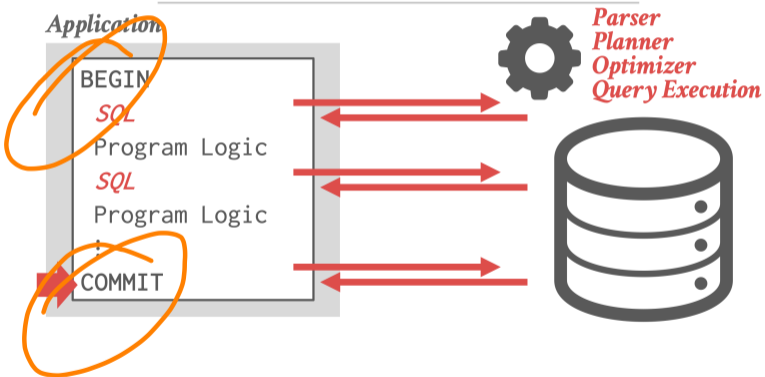
Conversational Database API



Conversational Database API



Conversational Database API



Conversational Database API

- The application has a "conversation" with the DBMS to store/retrieve data.
- Locks are held for the duration of the transaction
- Multiple network round-trips

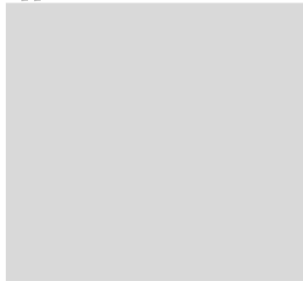
Embedded Database Logic

imperative +

- Move application logic into the DBMS to avoid multiple network round-trips and to extend the functionality of the DBMS.
- Potential Benefits
 - ▶ Efficiency
 - ▶ Reuse logic across web and mobile applications

Embedded Database Logic: Stored Procedures

Application



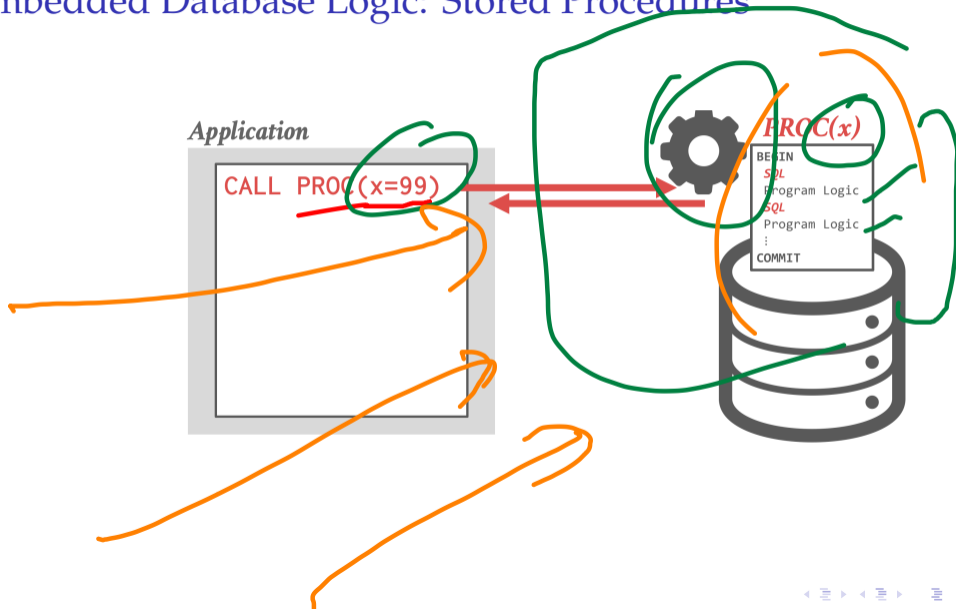
PROC(x)

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

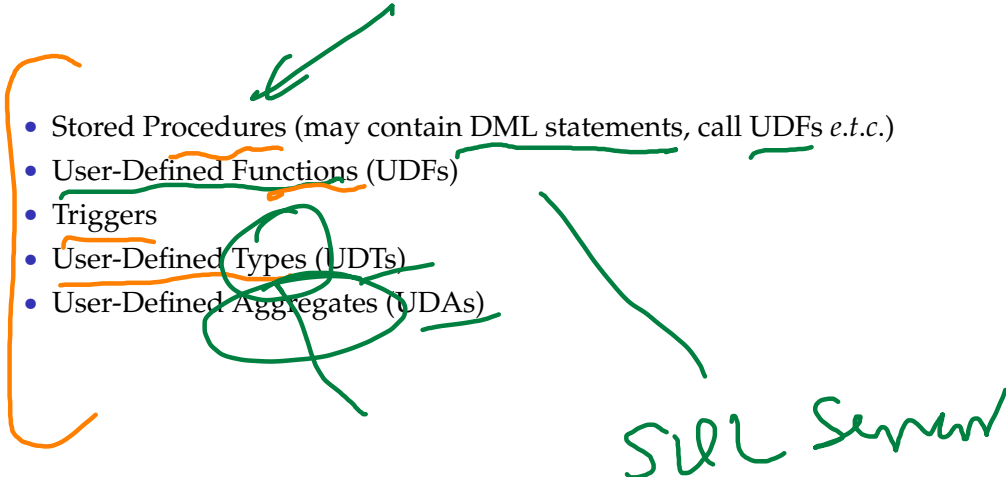


Stored Procedure

Embedded Database Logic: Stored Procedures



Embedded Database Logic

- Stored Procedures (may contain DML statements, call UDFs *e.t.c.*)
 - User-Defined Functions (UDFs)
 - Triggers
 - User-Defined Types (UDTs)
 - User-Defined Aggregates (UDAs)
- 

SQL Server

User-Defined Functions

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)
- Examples: PL/SQL, pLPG/SQL

SQLPGM

(Table-valued Functions)

UDF Example

- 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
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
```

Handwritten annotations: A red circle highlights 'c_custkey' in the SELECT statement. Green lines connect 'c_custkey' to '@ckey' in the function definition, and '@total' to 'SUM(o_totalprice)'. Orange lines connect 'o_custkey=@ckey' to '1', '2', and '3' written below. A red line underlines 'regular' in the IF-ELSE statement.

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.

imperative — declarative — functional — logic

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.

TLP
DLP

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.

proj filter

UDF Performance

TPC-H

```

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

```

--- User Defined Function

10000 times

UDF Performance

```
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
```

$z = f(y)$

$y \uparrow : z \uparrow$

index

UDF Performance

- Microsoft SQL Server
- TPC-H Q12 using a UDF (Scale Factor=1).
- **Reference**
 - ▶ Original Query: 0.8 sec
 - ▶ Query + UDF: 13 hr 30 min

Microsoft

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 Cray Lab.
- 2018 – Froid added to SQL Server 2019.

UDF In-lining

Froid: UDF In-lining

- 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.

- Reference

UDF → Sub-queries

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.

CTE



Sub-Queries – De-correlate

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

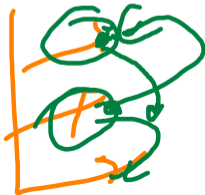
```
SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2020-04-22'
```

Sailors
club

Lateral Join

SQL Server / Cross Apply

- Subqueries appearing in FROM can be preceded by the key word LATERAL.
- This allows them to reference columns provided by preceding FROM items.
- Without LATERAL, each subquery is evaluated independently and so cannot cross-reference any other FROM item.
- LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined.

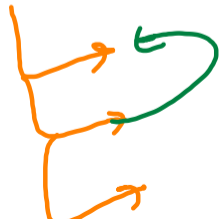
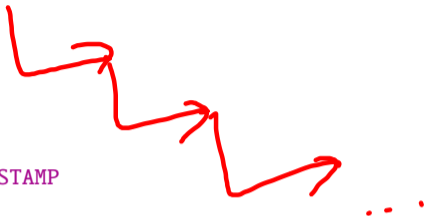


Lateral Join

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY, user_id INT, created TIMESTAMP
);
```

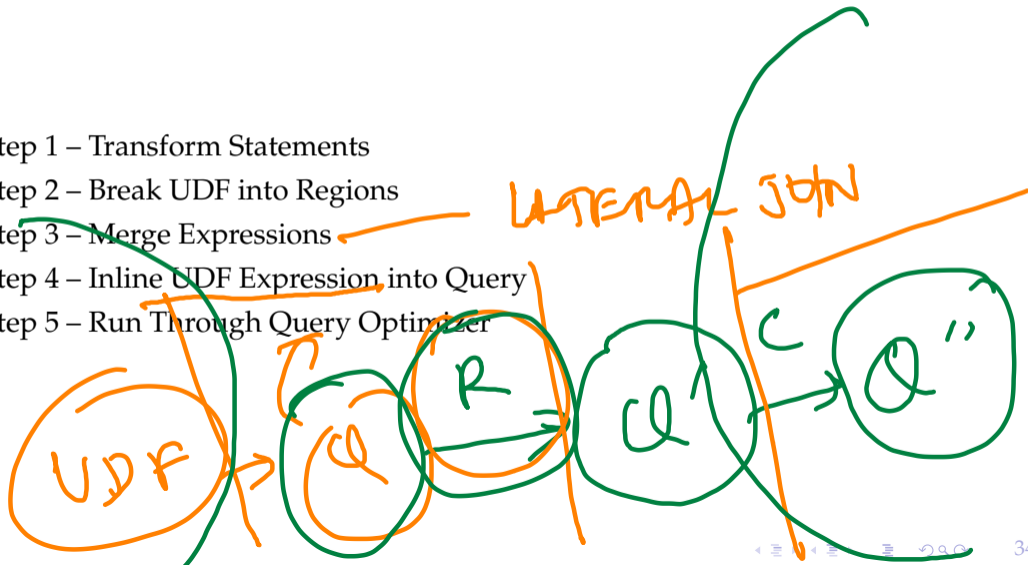
--- Query

```
SELECT user_id, first_order, next_order, id FROM
(SELECT user_id, min(created) AS first_order FROM orders GROUP BY user_id) 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)
o2 ON true LIMIT 1;
```

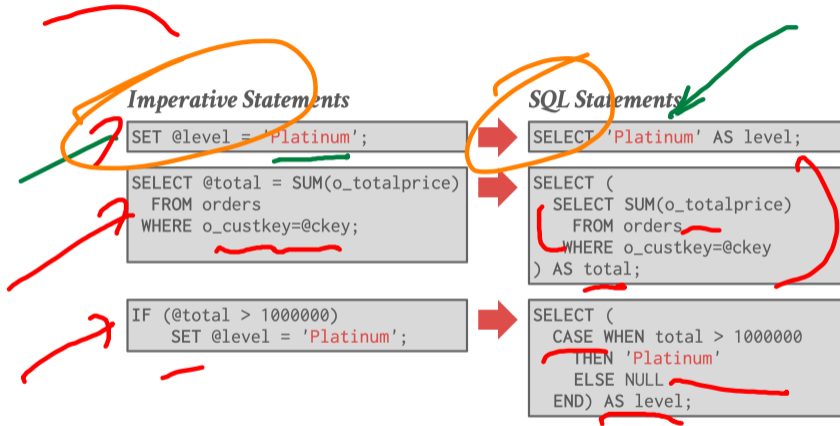


FROID Overview

- 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



Step 2 – Break UDF 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
```

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 UDF Expression into Query

Original Query

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



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

Inlining

Step 5 - Run Through Query Optimizer

```
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

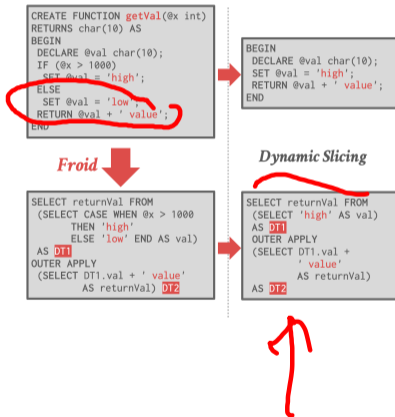
> 1000: high
< 1000: low

UDF

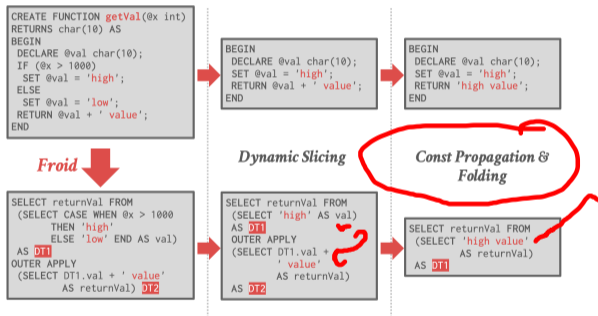
```
CREATE FUNCTION getVal(@x int)
RETURNS char(10) AS
BEGIN
  DECLARE @val char(10);
  IF (@x > 1000)
    SET @val = 'high';
  ELSE
    SET @val = 'low';
  RETURN @val + ' value';
END
```

```
SELECT getVal(5000);
```

Bonus Optimizations



Bonus Optimizations



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
```

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

Dynamic Slicing

```
SELECT returnVal FROM
(SELECT 'high' AS val)
AS DT1
OUTER APPLY
(SELECT DT1.val +
  ' value'
  AS returnVal)
AS DT2
```

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

Const Propagation & Folding

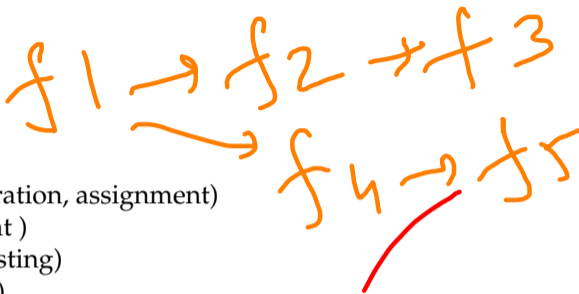
```
SELECT returnVal FROM
(SELECT 'high value'
  AS returnVal)
AS DT1
```

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

Dead Code Elimination

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

Supported Operations (2019)



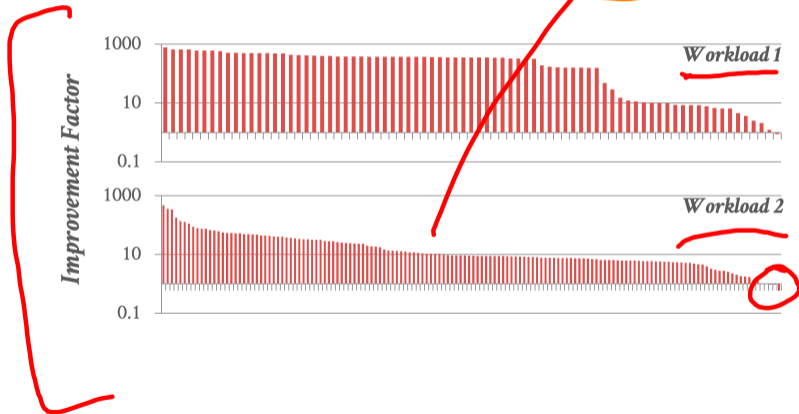
- T-SQL Syntax:
 - ▶ DECLARE, SET (variable declaration, assignment)
 - ▶ SELECT (SQL query, assignment)
 - ▶ IF / ELSE / ELSEIF (arbitrary nesting)
 - ▶ RETURN (multiple occurrences)
 - ▶ EXISTS, NOT EXISTS, ISNULL, IN, ... (Other relational algebra operations)
- UDF invocation (nested/recursive with configurable depth)
- All SQL datatypes.
- Limitations: Loops, Dynamic Queries, Exceptions

Applicability / Coverage

Workloads	Number of Scalar UDFs	Froid Compatible
Workload 1	178	150
Workload 2	90	82
Workload 3	22	21

cloud-native

UDF Improvement Study



UDFs to CTEs Conversion



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.
- Reference

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

Remove
CTEs

Step 1 – Static Single Assignment Form

```

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;

```

SSA

$$y \leftarrow x^n$$

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)
```

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)
  
```

Handwritten orange notes on the left side of the slide:

- A large orange arrow pointing from the left code block towards the right code block.
- Handwritten text "f₁" and "f₂" with arrows indicating a flow or relationship between the two code blocks.

Step 4 – WITH RECURSIVE

```

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

2 run(i1,p1,x,n) =
  let t0 = i1 >= n in
  if t0 then p1
  else

3 let p2 = p1 * x in
  let i2 = i1 + 1 in
  run(i2,p2,x,n)

```

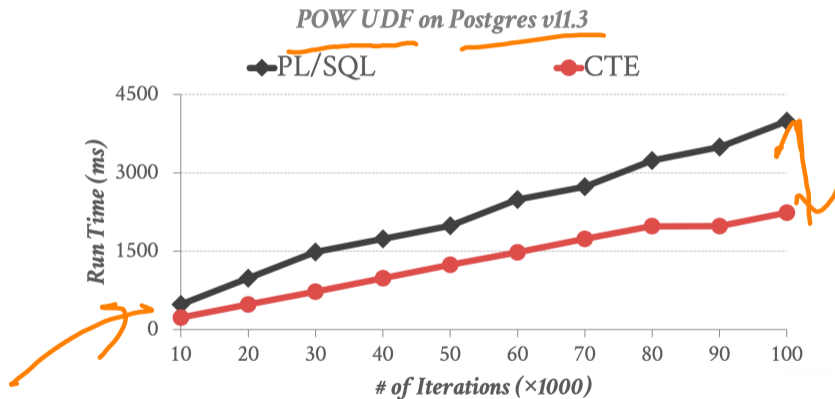
```

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;

```

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

UDFs-to-CTEs Evaluation



Conclusion

Parting Thoughts

- This is huge. You rarely get $500\times$ 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.

$f(1)$
 $f(2)$

Next Class

- Course Retrospective