TODAY’S AGENDA

Background
UDF In-lining
Working on Large Software Projects
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

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

\texttt{SQL}

Program Logic

\texttt{SQL}

Program Logic

\texttt{COMMIT}

Parser

Planner

Optimizer

Query Execution
EMBEDDED DATABASE LOGIC

Move application logic into the DBMS to avoid multiple network round-trips.

Potential Benefits
→ Efficiency
→ Reuse
APPLICATION

BEGIN

\textbf{SQL}

Program Logic

\textbf{SQL}

Program Logic

:\

COMMIT
EMBEDDED DATABASE LOGIC

Application

CALL PROC(x=99)

PROC(x)

BEGIN
SQL
Program Logic
SQL
Program Logic
:
COMMIT
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

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

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

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

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

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

Source: Karthik Ramachandra
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
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 2.14 ms for the system calls and 38.75 ms for the UDF. Thats a 19x increase.

```sql
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
```

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.

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.

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 Jim Gray Lab.

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 Jim 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-B.
2015 – Froid project begins @ MSFT Jim Gray Lab.
2018 – Froid added to SQL Server 2019.

Source: Karthik Ramachandra

Scalar UDF Inlining

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:

Source: Karthik Ramachandra

Carnegie Mellon University

Data Management System Laboratory
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
SUB-QUERIES – REWRITE

SELECT name FROM sailors AS S
WHERE EXISTS (
    SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
    AND R.day = '2019-03-25'
)

SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2019-03-25'
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 reference 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 @v = 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 v;
- 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
CREATE FUNCTION cust_level(@ckey int) RETURNS char(10) AS
BEGIN
    DECLARE @total float;
    DECLARE @level char(10);
    SELECT @total = SUM(o_totalprice) FROM orders WHERE o_custkey = @ckey;
    IF (@total > 1000000)
        SET @level = 'Platinum';
    ELSE
        SET @level = 'Regular';
    RETURN @level;
END

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

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

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

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

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

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

\[
\begin{align*}
\text{(SELECT NULL AS level,} \\
\text{(SELECT SUM(o_totalprice) } \\
\text{FROM orders} \\
\text{WHERE o_custkey=@ckey) AS total} \\
\text{)} \text{ AS E_R1}
\end{align*}
\]

\[
\begin{align*}
\text{(SELECT (} \\
\text{CASE WHEN E_R1.total > 1000000} \\
\text{THEN 'Platinum'} \\
\text{ELSE E_R1.level END) AS level} \\
\text{)} \text{ AS E_R2}
\end{align*}
\]

\[
\begin{align*}
\text{(SELECT (} \\
\text{CASE WHEN E_R1.total <= 1000000} \\
\text{THEN 'Regular'} \\
\text{ELSE E_R2.level END) AS level} \\
\text{)} \text{ AS E_R3}
\end{align*}
\]
STEP #3 – MERGE EXPRESSIONS

\[
\begin{align*}
&\text{(SELECT NULL AS level,} \\
&(\text{SELECT SUM(o_totalprice)} \\
&\quad \text{FROM orders} \\
&\quad \text{WHERE o_custkey=}@\text{ckey}) \text{ AS total} \\
&\quad ) \text{ AS E_R1} \\
\end{align*}
\]

\[
\begin{align*}
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total } > 1000000 \\
&\quad \text{THEN 'Platinum' } \\
&\quad \text{ELSE E_R1.level END) AS level} \\
&\quad ) \text{ AS E_R2} \\
\end{align*}
\]

\[
\begin{align*}
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total } \leq 1000000 \\
&\quad \text{THEN 'Regular' } \\
&\quad \text{ELSE E_R2.level END) AS level} \\
&\quad ) \text{ AS E_R3} \\
\end{align*}
\]

\[
\begin{align*}
&\text{SELECT E_R3.level FROM} \\
&(\text{SELECT NULL AS level,} \\
&(\text{SELECT SUM(o_totalprice)} \\
&\quad \text{FROM orders} \\
&\quad \text{WHERE o_custkey=}@\text{ckey}) \text{ AS total} \\
&\quad ) \text{ AS E_R1} \\
&\text{CROSS APPLY} \\
&(\text{SELECT (} \\
&\quad \text{CASE WHEN E_R1.total } > 1000000 \\
&\quad \text{THEN 'Platinum' } \\
&\quad \text{ELSE E_R1.level END) AS level} \\
&\quad ) \text{ AS E_R2} \\
&\text{CROSS APPLY} \\
&(\text{SELECT (} \\
&\quad \text{CASE WHEN E_R1.total } \leq 1000000 \\
&\quad \text{THEN 'Regular' } \\
&\quad \text{ELSE E_R2.level END) AS level} \\
&\quad ) \text{ AS E_R3} \\
\end{align*}
\]
STEP #3 – MERGE EXPRESSIONS

$$\begin{align*}
&\text{(SELECT } \text{NULL AS level,} \\
&\quad \text{(SELECT SUM(o_totalprice) FROM orders} \\
&\quad \quad \text{WHERE o_custkey=@ckey) AS total)} \text{ AS E_R1} \\
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total > 1000000 THEN 'Platinum' ELSE E_R1.level END) AS level)} \text{ AS E_R2} \\
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total <= 1000000 THEN 'Regular' ELSE E_R2.level END) AS level)} \text{ AS E_R3};
\end{align*}$$

$$\begin{align*}
&\text{(SELECT } \text{NULL AS level,} \\
&\quad \text{(SELECT SUM(o_totalprice) FROM orders} \\
&\quad \quad \text{WHERE o_custkey=@ckey) AS total)} \text{ AS E_R1} \\
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total > 1000000 THEN 'Platinum' ELSE E_R1.level END) AS level)} \text{ AS E_R2} \\
&\text{(SELECT (} \\
&\quad \text{CASE WHEN E_R1.total <= 1000000 THEN 'Regular' ELSE E_R2.level END) AS level)} \text{ AS E_R3};
\end{align*}$$
STEP #4 – INLINE EXPRESSION

Original Query

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

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

Inline Expression

```
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
```

**Modified Query**

```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 
    ) AS E_R3; 
) 
FROM customer;
```
STEP #5 - OPTIMIZE

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

---

```sql
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
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
**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**

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

**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)
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**

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

**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

<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

Source: Karthik Ramachandra
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.
ANDY’S
LIFE LESSONS
FOR WORKING
ON CODE
I have worked on a few large projects in my lifetime (2.5 DBMSs, 1 distributed system).
I have also read a large amount of “enterprise” code for legal stuff over multiple years.

But I’m not claiming to be all knowledgeable in modern software engineering practices.
OBSERVATION

Most software development is never from scratch. You will be expected to be able to work with a large amount of code that you did not write.

Being able to independently work on a large code base is the #1 skill that companies tell me they are looking for in students they hire.
PASSIVE READING

Reading the code for the sake of reading code is (usually) a waste of time. → It’s hard to internalize functionality if you don’t have direction.

It’s important to start working with the code right away to understand how it works.
TEST CASES

Adding or improving tests allows you to improve the reliability of the code base without the risk of breaking production code.
→ It forces you to understand code in a way that is not possible when just reading it.

Nobody will complain (hopefully) about adding new tests to the system.
REFACTORING

Find the general location of code that you want to work on and start cleaning it up.
→ Add/edit comments
→ Clean up messy code
→ Break out repeated logic into separate functions.

Tread lightly though because you are changing code that you are not familiar with yet.
Beyond working on the code, there will also be an established protocol for software development.

More established projects will have either training or comprehensive documentation.

→ If the documentation isn’t available, then you can take the initiative and try to write it.
PROJECT #3 SCHEDULE

**Status Meeting:** Next Week

**Status Update Presentation:** Monday April 8\textsuperscript{th}

**First Code Review:** Monday April 8\textsuperscript{th}
NEXT CLASS

Hash Tables!
Hash Functions!
Hash Joins!