Lecture #11: Server-side Logic Execution

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Why have Database Logic Embedded inside DBMS?

If we want to do some complicated logic based on data, we have 2 options. Either we can query data from
the SQL engine and process the data, and then query again.
This leads to multiple round-trips of data, which is very costly as seen in the result set serialization paper [3].
The computation will be based on the latest data rather than on stale snapshots.
The solution is to move application logic into the DBMS to avoid multiple network round-trips and to extend
the functionality of the DBMS. Potential Benefits: Efficiency and Reuse.
There are several different types of Embedded Database Logic, including User-Defined Functions (UDFs),
Stored Procedures, Triggers, User-Defined Types (UDTs), and User-Defined Aggregates (UDAs).

2 What are 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 input arguments (scalars), performs some computation, and then returns a result (scalars or tables).
Stored Procedures can be invoked outside a SQL query. Some DBMS make Stored Procedures unable to
update the database.

3 What languages are UDFs written in?

- SQL/PSM - SQL Standard
- PL/SQL - Oracle/DB2
- PL/pgSQL - PostgreSQL
- SQL PL - DB2
- Transact-SQL - MySQL/Sybase

Fewer network round-trips between the application server and DBMS are required for complex operations.
Some types of application logic are easier to express and read as UDFs than in SQL.
What are UDFs good for?

- UDFs 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.

Why are UDFs not being used as much anymore?

They can be slow due to the following reasons:

- Query optimizers treat UDFs as black boxes. The cost cannot be calculated if you don’t know what a UDF is going to do when you run it. This makes it difficult to have an optimal query plan, which relies heavily on cost models.
- It is difficult to parallelize UDFs due to correlated queries inside them. Furthermore, due to correlated queries inside the UDF, some DBMSs will only execute queries with a single thread.
- "Row By Agonizing Row" (RBAR): Some UDFs incrementally construct queries by processing each tuple sequentially and independently, which is a huge loss for analytics workloads. Things get even worse if the 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.

How to make UDFs run faster?

- **Compilation:** Compile interpreted UDF code into native machine code.
- **Parallelization:** Use user-defined annotations to figure out which portions of a UDF can be run in parallel.
7 Froid uses UDF Inlining

Froid automatically converts UDFs into relational expressions that are inlined as sub-queries [4]. This approach does not require the app developer to change the UDF code. The conversion is performed during the rewrite phase to avoid having to change the cost-based optimizer. Commercial DBMSs already have powerful transformation rules for executing sub-queries efficiently. Froid has five main steps:

- Transform Statements - Transform PL statements to SQL queries

![Figure 3: Step 1: Transform Statements](image)
• Break UDF into Regions - This allows reasoning about the contents and understanding the dependencies between those regions, which are then expressed as lateral joins.

Figure 4: Step 2: Break into Regions

• Merge Expressions - Combine multiple expressions into one region and link them together with lateral joins.

Figure 5: Step 3: Merge Expressions

• Inline UDF Expression into Query - Embed the query into the main query.
• Run Through Query Optimizer - Now both the main query and the UDF query can be optimized together.

Other Optimizations in the Transformation Process

We also receive multiple benefits from normal code optimizations, such as:
**STEP #4: INLINE EXPRESSION**

Original Query

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

SELECT c_custkey, (SELECT E.R3.level
(SELECT NUL AS level,
(SELECT SUM(o_totalprice)
FROM orders
WHERE o_custkey=c_key) AS total
) AS c.R1
CROSS APPLY
(SELECT (CASE WHEN E.R1.total > 1000000
THEN 'Platinum'
ELSE E.R1.level END) AS level
) AS c.R2
CROSS APPLY
(SELECT (CASE WHEN E.R1.total <= 1000000
THEN 'Regular'
ELSE E.R2.level END) AS level
) AS c.R3;
) FROM customer;

Figure 6: Step 4: Inline Expression

**STEP #5: OPTIMIZE**

```sql
SELECT c_custkey, (SELECT NULL AS level,
(SELECT SUM(o_totalprice)
FROM orders
WHERE o_custkey=c_key) AS total
) AS c.R1
CROSS APPLY
(SELECT (CASE WHEN E.R1.total > 1000000
THEN 'Platinum'
ELSE 'Regular'
END) AS level
) AS c.R2
CROSS APPLY
(SELECT (CASE WHEN E.R1.total <= 1000000
THEN 'Regular'
ELSE E.R2.level END) AS level
) AS c.R3;
) FROM customer;
```

Figure 7: Step 5: Optimize
• Dynamic Slicing: Identifying the relevant statements that affect a given point of interest during pro-
gram execution.
• Constant Propagation and Folding: Replacing expressions with their constant values to simplify the
code.
• Dead Code Elimination: Removing unreachable or unused code to improve efficiency.

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. The two approaches are:
• Rewrite to de-correlate and/or flatten them.
• Decompose nested query and store result in a temporary table. Then perform outer joins with the
temporary table.

Lateral Join
A lateral inner subquery can refer to fields in rows of the table reference to determine which rows to return.
This allows you to have sub-queries in the FROM clause. The DBMS iterates through each row in the
referenced table 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.

8 APFEL Uses UDFs-to-CTEs Conversion
Rewrite UDFs into plain SQL commands [1]. Utilize 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. The five main steps to convert UDFs to
CTEs are:
• Static Single Assignment Form - Define each variable once with multiple labels and goto.
• Administrative Normal Form - The last line of each function calls another function.
• Mutual to Direct Recursion - Call functions happen in only one direction.
• Tail Recursion to WITH RECURSIVE - Convert to SQL Query having WITH RECURSIVE.
• Run Through Query Optimizer - Normal SQL optimization process.

9 UDF Batching
• Transform UDF statements into UPDATE queries that operate on a temporary table representing
the state of variables in the UDF.
• It has been shown that batching works better than inlining [2].
References


