Intro to Database Systems (15-445/645)

24 Application Logic Inside the DB

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Homework 5 ongoing
→ Due Friday, April 21st at 11:59 p.m.

Project 4 ongoing
→ Due Friday, April 28th at 11:59 p.m.

Guest lecture on Monday is in Zoom
→ https://bit.ly/3KPmJLI

Final exam Monday, May 1st, 8:30 – 11:30 a.m.

LAST TIME

CAP theorem

Distributed OLAP for decision support
→ Various topics
→ Distributed join algorithms

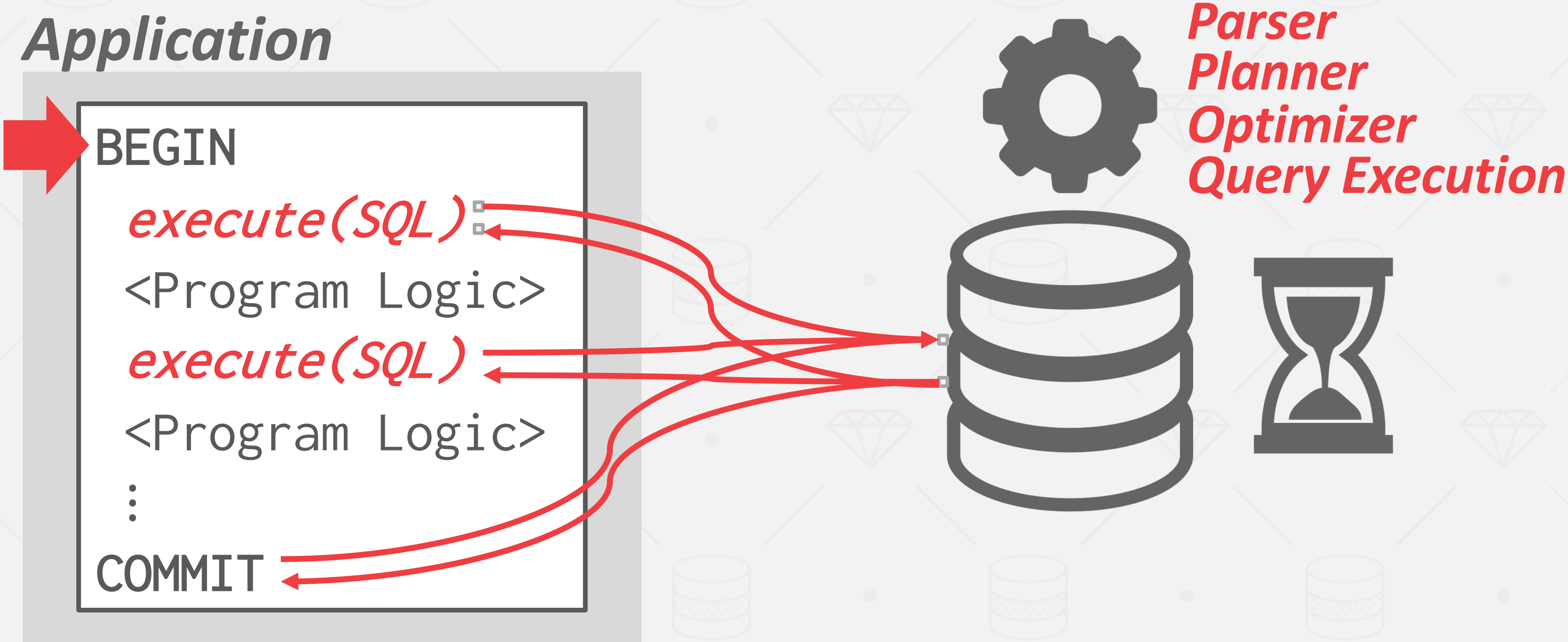
OBSERVATION

Until now, we have assumed that all the logic for an application is external to the database.

The application has a "conversation" with the DBMS to store/retrieve data.

→ Each DBMS has its own network protocol.
→ Client-side APIs: JDBC, ODBC

CONVERSATIONAL DATABASE API



APPLICATION LOGIC IN THE DATABASE

Moving application logic into the DBMS can (potentially) provide several benefits:

→ Fewer network round-trips.
→ Immediate notification of changes.
→ DBMS spends less time waiting during transactions.
→ Developers do not have to reimplement functionality.

TODAY'S AGENDA

User-defined Functions
Stored Procedures
Triggers
Change Notifications
User-defined Types
Views

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 DEFINITION

Return Types:

→ Scalar Functions: Return a single data value

→ Table Functions: Return a single result table.

Computation Definition:

→ SQL Functions

→ External Programming Language

UDF – SQL FUNCTIONS

A SQL-based UDF contains a list of queries that the DBMS executes in order when invoked.

→ The function returns the result of the last query executed.

```
CREATE FUNCTION get_foo(int)
  RETURNS foo
  LANGUAGE SQL AS $$
  SELECT * FROM foo WHERE foo.id = $1;
$$;
```

Input Args (`get_foo(int)`)

Return Args (`RETURNS foo`)

Function Body (`SELECT * FROM foo WHERE foo.id = $1;`)

```
SELECT get_foo(1);
```

```
SELECT * FROM get_foo(1);
```

UDF – SQL FUNCTIONS

SQL Standard provides the ATOMIC keyword to tell the DBMS that it should track dependencies between SQL UDFs.

```
CREATE FUNCTION get_foo(int)
  RETURNS foo
  LANGUAGE SQL
  BEGIN ATOMIC;
    SELECT * FROM foo WHERE foo.id = $1;
  END;
```

UDF – EXTERNAL PROGRAMMING LANGUAGE

Some DBMSs support writing UDFs in languages other than SQL.
→ **SQL Standard**: SQL/PSM
→ **Oracle/DB2**: PL/SQL
→ **Postgres**: PL/pgSQL
→ **MSSQL/Sybase**: Transact-SQL

Other systems support more common programming languages:
→ Sandbox vs. non-Sandbox

PL/PGSQL UDF EXAMPLE (1)

```
CREATE OR REPLACE FUNCTION get_foo(int)
  RETURNS SETOF foo
  LANGUAGE plpgsql AS $$
  BEGIN
    RETURN QUERY
       SELECT * FROM foo WHERE foo.id = $1;
  END;
$$;
```

PL/PGSQL UDF EXAMPLE (2)

```
CREATE OR REPLACE FUNCTION sum_foo(i int)
  RETURNS int AS $$
  DECLARE foo_rec RECORD;
  DECLARE out INT;
  BEGIN
    out := 0;
    FOR foo_rec IN SELECT id FROM foo
                        WHERE id > i LOOP
      out := out + foo_rec.id;
    END LOOP;
    RETURN out;
  END;
$$ LANGUAGE plpgsql;
```

Variable Declaration

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

```
SELECT l_shipmode,
       SUM(CASE
           WHEN o_orderpriority <
           THEN 1 ELSE 0 END
       ) AS low_line_count
  FROM orders, lineitem
 WHERE o_orderkey = l_orderkey
   AND l_shipmode IN ('MAIL','SHI
   AND l_commitdate < l_receiptda
   AND l_shipdate < l_commitdate
   AND l_receiptdate >= '1994-01
   AND dbo.cust_name(o_custkey)
 GROUP BY l_shipmode
 ORDER BY l_shipmode
```

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: Karthik Ramachandra

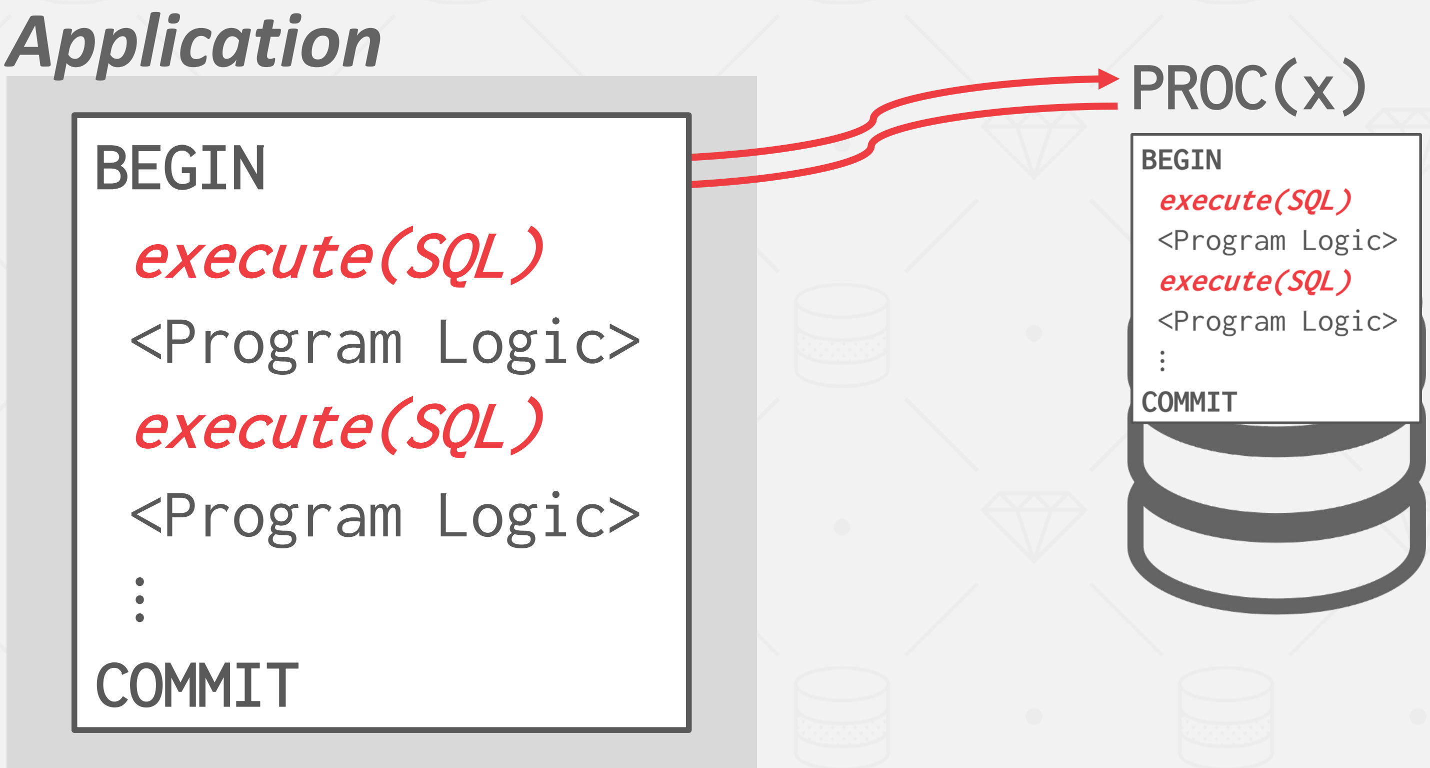
STORED PROCEDURES

A **stored procedure** is a self-contained function that performs more complex logic inside of the DBMS.

→ Can have many input/output parameters.
→ Can modify the database table/structures.
→ Not normally used within a SQL query.

Some DBMSs distinguish UDFs vs. stored procedures, but not all.

STORED PROCEDURES



STORED PROCEDURE EXAMPLE

```
CREATE OR REPLACE PROCEDURE transfer(sender INT, receiver INT, amount FLOAT)
  LANGUAGE plpgsql AS $$
    DECLARE sndr_bal INT;
    DECLARE sndr_name VARCHAR;
    BEGIN
      SELECT name, balance INTO sndr_name, sndr_bal
        FROM accounts WHERE id = sender;
      IF sndr_bal < amount THEN
        RAISE EXCEPTION '% does not have enough money!', sndr_name;
      END IF;
      UPDATE accounts SET balance = balance - amount WHERE id = sender;
      UPDATE accounts SET balance = balance + amount WHERE id = receiver;
      COMMIT;
    END;
$$;
```

```
CALL transfer(1, 2, 50);
```

STORED PROCEDURE VS. UDF

A UDF is meant to perform a subset of a read-only computation within a query.

A stored procedure is meant to perform a complete computation that is independent of a query.

DATABASE TRIGGERS

A **trigger** instructs the DBMS to invoke a UDF when some event occurs in the database.

The developer has to define:

→ What type of **event** will cause it to fire.
→ The **scope** of the event.
→ When it fires **relative** to that event.

TRIGGER EXAMPLE

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE TABLE foo_audit (
    id SERIAL PRIMARY KEY,
    foo_id INT REFERENCES foo (id)
```

```
CREATE OR REPLACE FUNCTION log_foo_updates()
                   RETURNS trigger AS $$
  BEGIN
   IF NEW.val <> OLD.val THEN
     INSERT INTO foo_audit
                  (foo_id, orig_val, cdate)
           VALUES (OLD.id, OLD.val, NOW());
$$ LANGUAGE plpgsql;
```

Tuple Versions

```
CREATE TRIGGER foo_updates
   BEFORE UPDATE ON foo FOR EACH ROW
  EXECUTE PROCEDURE log_foo_updates();
```

TRIGGER DEFINITION

Event Type:
- → INSERT
- → UPDATE
- → DELETE
- → TRUNCATE
- → CREATE
- → ALTER
- → DROP

Event Scope:
- → TABLE
- → DATABASE
- → VIEW
- → SYSTEM

Trigger Timing:
- → Before the query executes.
- → After the query executes
- → Before each row the query affects.
- → After each row the query affects.
- → Instead of the query.

CHANGE NOTIFICATIONS

A **change notification** is like a trigger except that the DBMS sends a message to an external entity that something notable has happened in the database.

→ Think a "pub/sub" system.

→ Can be chained with a trigger to pass along whenever a change occurs.

SQL standard: `LISTEN` + `NOTIFY`

NOTIFICATION EXAMPLE

```
CREATE OR REPLACE FUNCTION notify_foo_updates()
                   RETURNS trigger AS $$
  DECLARE notification JSON;
  BEGIN
    notification = row_to_json(NEW);
    PERFORM pg_notify('foo_update',
                      notification::text);
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Notification Payload

```
CREATE TRIGGER foo_notify
  AFTER INSERT ON foo_audit FOR EACH ROW
  EXECUTE PROCEDURE notify_foo_updates();
```

OBSERVATION

All DBMSs support the basic primitive types in the SQL standard. They also support basic arithmetic and string manipulation on them.

But what if we want to store data that doesn't match any of the built-in types?

```
coordinate (x, y, label)
```

COMPLEX TYPES

Approach #1: Attribute Splitting

→ Store each primitive element in the complex type as its own attribute in the table.

Approach #2: Application Serialization

→ Java serialize, Python pickle
→ Google Protobuf, Facebook Thrift
→ JSON / XML

```
INSERT INTO locations
   (x, y, label)
VALUES
   (10, 20, "OTB");
```

```
CREATE TABLE locations (
   coord JSON NOT NULL
);
```

```
INSERT INTO location (coord)
VALUES (
 '{x:10, y:20, label:"OTB"}'
);
```

USER-DEFINED TYPES

A **user-defined type** is a special data type that is defined by the application developer that the DBMS can stored natively.

→ First introduced by Postgres in the 1980s.
→ Added to the SQL:1999 standard as part of the "object-relational database" extensions.

Sometimes called **structured user-defined types** or **structured types**.

USER-DEFINED TYPES

Each DBMS exposes a different API that allows you to create a UDT.

→ Postgres/DB2 supports creating composite types using built-in types.
→ Oracle supports PL/SQL.
→ MSSQL/Postgres only support type definition using external languages (.NET, C)

```
CREATE TYPE coordinates AS (
  x INT, y INT, label VARCHAR(32)
);
```

PostgreSQL

```
CREATE TYPE coordinates AS OBJECT (
  x INT NOT NULL,
  y INT NOT NULL,
  label VARCHAR(32) NOT NULL
);
```

ORACLE

VIEWS

Creates a "virtual" table containing the output from a SELECT query. The view can then be accessed as if it was a real table.

This allows programmers to simplify a complex query that is executed often.

→ It won't make the DBMS magically run faster though.

Often used as a mechanism for hiding a subset of a table's attributes from certain users.

VIEW EXAMPLE (1)

Create a view of the CS student records with just their id, name, and login.

```
CREATE VIEW cs_students AS
  SELECT sid, name, login
    FROM student
   WHERE login LIKE '%@cs';
```

Original Table

sid	name	login	age	gpa
53666	RZA	rza@cs	53	3.5
53677	Justin Bieber	jb@ece	23	2.25
53688	Tone Loc	tloc@mld	56	3.8
53699	Andy Pavlo	pavlo@cs	41	3.0

```
SELECT * FROM cs_students;
```

sid	name	login
53666	RZA	rza@cs
53699	Andy Pavlo	pavlo@cs

VIEW EXAMPLE (2)

Create a view with the average age of all of the students.

```
CREATE VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

VIEWS VS. SELECT INTO

VIEW
→ Dynamic results are only materialized when needed.

SELECT…INTO
→ Creates static table that does not get updated when student gets updated.

```
CREATE VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

```
SELECT AVG(gpa) AS avg_gpa
  INTO cs_gpa
  FROM student
 WHERE login LIKE '%@cs';
```

UPDATING VIEWS

The SQL-92 standard specifies that an application is allowed to modify a VIEW if it has the following properties:

→ It only contains one base table.
→ It does not contain grouping, distinction, union, or aggregation.

MATERIALIZED VIEWS

Creates a view containing the output from a SELECT query that is retained (i.e., not recomputed each time it is accessed).

→ Some DBMSs automatically update matviews when the underlying tables change.

→ Other DBMSs (PostgreSQL) require manual refresh.

```
CREATE MATERIALIZED VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

CONCLUSION

Moving application logic into the DBMS has lots of benefits.
→ Better efficiency
→ Reusable across applications

But it has problems:
→ Not portable
→ DBAs don't like constant change.
→ Potentially need to maintain different versions.