Embedded Database Logic
PROJECT #2 is due Wednesday October 25th @ 11:59am

Exams are available to view during my office hours.
→ Bring your CMU ID.
→ You are not allowed to take it with you.
→ Take a photo of any page that you want regraded.
EXTRA CREDIT (OPTIONAL)

We are building an open-source database benchmarking framework.

Pick a DBMS that you want to try.
→ Must support transactions
→ Must support SQL/JDBC

Get a benchmark to run on it.
Get 10% extra credit.

http://oltpbenchmark.com
DATABASE TALKS

Oracle In-Memory Database
→ Tuesday Oct 24th @ 6pm
→ GHC 6115

Battery Ventures Talk
→ Wednesday Oct 25th @ 4:30pm
→ GHC 4405

Kdb Time-series DB Talk
→ Thursday @ Oct 26th @ 12:00pm
→ CIC 4th Floor
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

\textcolor{red}{SQL}

Program Logic

\textcolor{red}{SQL}

Program Logic

; COMMIT
CONVERSATIONAL DATABASE API

Application

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

Parser
Planner
Optimizer
Query Execution
EMBEDDED DATABASE LOGIC

What if we could move complex application logic into the DBMS to avoid multiple network round-trips?

Potential Benefits

→ Efficiency
→ Reuse
TODAY'S AGENDA

User-defined Functions
Stored Procedures
Triggers
Change Notifications
User-defined Types
Views
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
CREATE TABLE foo (  
id INT PRIMARY KEY,  
val VARCHAR(16)  
);

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

UDF – SQL FUNCTIONS

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

→ The function returns whatever the result is of the last query executed;
UDF – SQL FUNCTIONS

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

→ The function returns whatever the result is of the last query executed;

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

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

→ The function returns whatever the result is of the last query executed;

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

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

Other systems support more common programming languages:

→ Sandbox vs. non-Sandbox
CREATE OR REPLACE FUNCTION get_foo(int)
  RETURNS SETOF foo AS $$
BEGIN
  RETURN QUERY SELECT * FROM foo
    WHERE foo.id = $1;
END;
$$ LANGUAGE plpgsql;
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;
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.
STORED PROCEDURES

Application

BEGIN
  SQL
  Program Logic
  SQL
  Program Logic :
  COMMIT
CALL PROC(x=99)

PROC(x)
BEGIN
SQL
Program Logic
SQL
Program Logic
:
COMMIT
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 DEFINITION

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

Event Scope:  
→ TABLE  
→ DATABASE  
→ VIEW  
→ SYSTEM

Trigger Timing:  
→ Before the statement executes.  
→ After the statement executes  
→ Before each row that the statement affects.  
→ After each row that the statement affects.  
→ Instead of the statement.
CREATE TABLE foo (
    id INT PRIMARY KEY,
    val VARCHAR(16)
);

CREATE TABLE foo_audit (
    id SERIAL PRIMARY KEY,
    foo_id INT REFERENCES foo (id),
    orig_val VARCHAR,
    cdate TIMESTAMP
);
CREATE TABLE foo (id INT PRIMARY KEY, val VARCHAR(16));

CREATE TABLE foo_audit (id SERIAL PRIMARY KEY, foo_id INT REFERENCES foo (id), orig_val VARCHAR, cdate TIMESTAMP);

CREATE OR REPLACE FUNCTION log_foo_updates() RETURNS trigger AS $$
BEGIN
  IF NEW.val <> OLD.val THEN
    INSERT INTO foo_audit (foo_id, orig_val, cdate) VALUES (OLD.id, OLD.val, NOW());
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
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),
    orig_val VARCHAR(16),
    cdate TIMESTAMP
);

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

CREATE TRIGGER foo_updates
BEFORE UPDATE ON foo
FOR EACH ROW
EXECUTE PROCEDURE log_foo_updates();
```
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**
CREATE OR REPLACE FUNCTION notify_foo_updates()
    RETURNS trigger AS $$
DECLARE notification JSON;
BEGIN
    IF NEW.val <> OLD.val THEN
        notification = row_to_json(NEW);
        PERFORM pg_notify('foo_update',
                           notification::text);
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER foo_notify
    BEFORE UPDATE ON foo
    FOR EACH ROW
    EXECUTE PROCEDURE notify_foo_updates();
OBSERVATION

All DBMSs or less support all of the 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?
DATA SERIALIZATION

One potential solution is to just store the complex in its serialized form:
→ Java serialize, Python pickle
→ Google Protobuf, Facebook Thrift
→ JSON / XML

This has problems:
→ How do you edit a sub-element?
→ How does the optimizer estimate selectivity on predicates that access serialized data?
→ How do you execute aggregates and other functions on the serialized object?
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.
→ Oracle supports PL/SQL.
→ DB2 supports creating types based on built-in types.
→ MSSQL/Postgres only support type definition using external languages (.NET, C)
VIEWS

Creates a "virtual" table containing the output from a **SELECT** query.

Mechanism for hiding data from view of certain users.

Can be used to simplify a complex query that is executed often.
→ Won’t make it faster though!
Create a view of the CS student records with just their id, name, and login.

**Original Table**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kanye West</td>
<td>kw@cs</td>
<td>40</td>
<td>3.5</td>
</tr>
<tr>
<td>53677</td>
<td>Justin Bieber</td>
<td>jb@ece</td>
<td>23</td>
<td>2.25</td>
</tr>
<tr>
<td>53688</td>
<td>Tone Loc</td>
<td>tloc@isr</td>
<td>51</td>
<td>3.8</td>
</tr>
<tr>
<td>53699</td>
<td>Andy Pavlo</td>
<td>pavlo@cs</td>
<td>36</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**View**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kanye West</td>
<td>kw@cs</td>
</tr>
<tr>
<td>53699</td>
<td>Andy Pavlo</td>
<td>pavlo@cs</td>
</tr>
</tbody>
</table>
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.
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.
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
→ PL/SQL is awkward to write.
→ DBAs don't like constant change.
→ Potentially need to maintain different versions.
NEXT CLASS

TRANSACTIONS!!!