Lecture #15: Embedded Database Logic

15-445/645 Database Systems (Fall 2018)
https://15445.courses.cs.cmu.edu/fall2018/
Carnegie Mellon University
Prof. Andy Pavlo

1 Motivation

Until now, we have assumed that all of the logic for an application is located in the application itself. Most applications interact with the DBMS using a "conversational" API (e.g., JDBC, ODBC). This is where the application sends a query request to the DBMS and then waits for a response. After the DBMS sends a response, it then waits for the next request from the application for that connection.

It may be possible to move complex application logic into the DBMS to avoid multiple network round-trips. Doing this can improve efficiency and reusability in the application.

The downside of these methods is that the syntax is often not portable across different DBMSs. And depending on the engineering practices of an organization, you may need to maintain different versions of the embedded database logic.

2 User-Defined Functions

A *user defined function* is a function written by the application developer that extends the system's functionality beyond its built-in operations. Each function takes in scalar input arguments, performs some computation, and then returns a result (scalar, table). A UDF can only be invoked as part of a SQL statement.

Return Types:

- **Scalar Functions:** Return a single data value.
- **Table Functions:** Return a single result table.

Function Body:

- **SQL Functions:** A SQL-based UDF contains a list of SQL statements that the DBMS executes in order when the UDF is invoked. The UDF returns whatever the result is of the last query.
- **Native Programming Language:** The developer can write a UDF in a language that is natively supported by the DBMS. Examples: `SQL/PSM` (SQL Standard), `PL/SQL` (Oracle, DB2), `PL/pgSQL` (Postgres), `Transact-SQL` (MSSQL/Sybase).
- **External Programming Language:** UDFs written in more conventional programming languages (e.g., C, Java, JavaScript, Python) run a separate process (i.e., sandbox) to prevent them from crashing the DBMS process.

3 Stored Procedures

A *stored procedure* is a self-contained function that performs more complex logic inside of the DBMS. Unlike a UDF, a stored procedure can be invoked on its own without having to be part of a SQL statement.

UDFs are also usually meant to be read-only, while stored procedures are allowed to modify the DBMS.

4 Triggers

A *trigger* instructs the DBMS to invoke a UDF when some event occurs in the database. Some examples of trigger usage are constraint checking or auditing any time a tuple is modified in a table.

Each trigger is defined with the following properties:

- **Event Type:** Type of modification (`INSERT`, `UPDATE`, `DELETE`, `ALTER`).
- **Event Scope:** Scope of the modification (`TABLE`, `DATABASE`, `VIEW`).
- **Timing:** When the trigger should be activated based on statement (before, after, instead of).

5 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. They can be chained with a trigger to pass along whenever a change occurs. Notifications are asynchronous, meaning that they are only pushed to listening connection whenever they interact with the DBMS. Some ORMs will poll the DBMS with lightweight "`SELECT 1`" every so often to retrieve new notifications.

Commands:

- `LISTEN`: The connection registers with the DBMS to listen for notifications at the named event queue.
- `NOTIFY`: Push a notification to any connection that is listening at named event queue. Syntax details vary per DBMS implementation.

6 User-Defined Types

Most DBMSs support the basic primitive types defined in the SQL standard (e.g., ints, floats, varchars). But sometimes that application wants to store complex types that are comprised of multiple primitive types. Or these complex types might have different behaviors for various arithmetic operators.

One potential solution is to store split the complex type and store each of its primitive element as its own attribute in the table. The problem with this is that you have to make sure that the application knows how to split/combine the complex type. Another solution is to let the application serialize the complex type (e.g., Java "serialize", Python "pickle", Google Protobufs) and store it as a blob in the database. The problem with this approach is that it not possible to edit sub-attributes in the type without first deserializing the entire blob. Likewise, the DBMS's optimizer is unable estimate selectivity on predicates that access serialized data.

A better approach is to use a *user-defined type* (UDT). This is a special data type that is defined by the application developer that the DBMS can be stored natively. Each DBMS exposes a different API that allows you to create a UDT. This allows you override basic operators and functions.

7 Views

A database *views* is "virtual" table that contains the output from a `SELECT` query. The view can then be accessed as if it was a real table. Under the hood, queries on views are converted into a single query using the original query that generated view. Views allow programmers to simplify a complex query that is executed often. It is often also used as a mechanism for hiding a subset of a table's attributes from certain users. One can only update a view if it only contains a single based table, and that it does not contain aggregations, distinctions, union, or grouping.

Unlike `SELECT...INTO`, a view does not allocate a table to store the result of the view. A *materialized view* maintains the result of a view internally that is automatically updated when the underlying tables change.