TABLE-DRIVEN PROGRAMMING IN SQL FOR ENTERPRISE INFORMATION SYSTEMS*

Hung-chih Yang
UCLA Computer Science Department
4732 Boelter Hall, Los Angeles, CA 90095, USA

D. Stott Parker
UCLA Computer Science Department
4732 Boelter Hall, Los Angeles, CA 90095, USA

Keywords: Table-Driven Programming, Object-Relational Databases, Dynamic Method Dispatch.

Abstract: In database systems, business logic is usually implemented in the forms of external processes, stored procedures, user-defined functions, components, objects, constraints, triggers, etc. In this paper, we advocate the idea of storing business logic – in the form of functions – as data in tables. This idea gives a basis for applying the software-engineering methodology of *table-driven programming* in SQL. The query evaluation process then needs only to be extended with mechanical evaluation of "joined" data and functions. This approach can make understanding and maintenance of stored business logic transparent as relational data. In short, data and functions are integrated in a relational manner. Using a common enterprise application as an example, we demonstrate this methodology with an existing ORDBMS capable of storing polymorphic objects. We also discuss this approach's shortcomings and alternatives.

1 INTRODUCTION

One of the authors once participated in a decision-support project that involved thousands of forecasting models. These models were straightforward mathematical formulas generated by a statistical software package and stored in a relational database. They were routinely joined with a set of data relations, then evaluated to produce important business forecasts. The formulas were originally created in text files, and a parser was used to separate coefficients, variables, and function calls from the formula structures. The separated formula components were then stored in several normalized relations in the forms of numbers, strings, and IDs. Formula structures were stored in a relation as BLOBs. An external forecasting process routinely joined several data relations with the formula relations and rebuilt forecasting formulas with plugged-in variable values. Forecast values were then computed in the external process and stored back in a relation. In order to reduce network communication, function relations and data relations were queried separately and a C++-implemented forecasting process had to simulate a nested loop **join** in order to put data and reconstructed functions together. This experience made us wonder: why can't a relational database store formulas in a relation directly, therefore formula tuples can be joined with data tuples to generate results?

In fact, this functionality that is missing in RDBMS is a programming methodology called *table-driven programming*. This methodology is popular in script programming and in parsers. Its basic principles are (a) storing data and instructions in tables, and (b) controlling programming logic by using conditions (or states) that select data and instruction out of the tables for execution. The mechanism of selecting data and instuctions is a general-purpose, straightforward, and mechanical process. Each individual part of programming logic can be easily replaced through updating values in tables. If only data is used to direct the program logic, then the term *data-driven* is a more precise description. For applying table-driven programming in SQL, relational databases could provide a vast repository for business logic and SQL queries then offer transparent evaluation of business logic.

In this paper we first discuss a methodology called *table-driven programming in SQL* and present a common enterprise database application that would greatly benefit from it. Later we discuss how to use

*This research was supported by NIH Grant 1P20MH065166-01, NIH Grant 1U54RR021813, and the UCLA *Center for Computational Biology (CCB)*, an NIH Center of Excellence.

Yang H. and Stott Parker D. (2005).
TABLE-DRIVEN PROGRAMMING IN SQL FOR ENTERPRISE INFORMATION SYSTEMS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 424-427
DOI: 10.5220/0002526004240427
Copyright © SciTePress

ORDBMS to implement table-driven applications and the drawbacks of this approach.

2 TABLE-DRIVEN PROGRAMMING IN SQL

One great benefit brought by relational databases is the clear representation of data and their relationships, but this benefit does not extend to traditional stored procedures and functions. Unlike relational tables, the imperative nature of stored procedures is navigational. Convoluted subroutines calls make a program hard to understand and maintain, and business logic can be hidden beyond recognition, while Object-Oriented (or Object-Relational) programming hides meandering navigational behavior behind communications among objects. In order to make business logic more transparent, we advocate the introduction of table-driven programming in relational databases. The goal is to transform the communications among functional modules from explicit navigations in stored procedures to declarative joins in a relational manner.

The design process of a table-driven system includes two stages: *factor* and *assemble* (or *join*). Since this two-stage process is to apply the relational model on programming code, we call it *relational programming* as well.

- **The Factor Stage**
  In this stage, a monolithic system of rules is split into simple rules and formulas that can be stored in attributes of normalized relations. After being stored in a relation, they can be easily located and manipulated just like other data. By contrast, maintaining formulas in a lengthy procedure like the one in figure 1 could be quite tedious and cumbersome.
- **The Assemble (or Join) Stage**
  In the assemble stage, users can write declarative queries to join tables of rules, formulas, and data together to find results. The query shown in figure 2 needs only join operations and evaluation of stored functions.

Notice that in both stages, we utilize only SQL constructs: functions of SQL expressions in the factor stage for storing granular business logic and SQL queries in the assemble stage for a table-driven computing process.

Developers may start a software project following the traditional database design and analysis process. During the design process, some of the relations may contain function attributes. Once a schema is ready, developers can build business logic working on queries and views that join data and functions together, rather than coding complicated imperative stored or external procedures. Individual rules and formulas can be easily replaced and redefined dynamically, while the overall picture of the business logic stays unchanged in easy-to-understand queries. In a traditional database design, the whole business logic and subroutines may require rewriting if there is any specification change (even a minor one) and only data can be dynamically changed (because data is defined relationally, not code).

Table 1: Schema for a bonus computation system

Employees

EmployeeID	EmployeeName	Department	BaseSalary

EmployeeBonus

EmployeeID	**BonusName**	BonusArgument

3 EXAMPLE: COMPUTING YEAR-END BONUS

Computing an employee's pay has become more and more complicated. Many kinds of deductions, reimbursements, direct transfers, and bonuses can be added to an employee's paycheck before or after tax. Also there are a lot of company and government regulations and ever-changing tax laws to consider. Worst of all, all these factors can differ from person to person because of the positions they have and the locations of their work, and those conditions are never static. A common way of dealing with this paycheck computing problem is to store numbers of rates, bonuses, deductions, and direct transfers in relations and use one or more stored or external procedures to query these relations and calculate the final paycheck amount. In order to achieve correctness, great scrutiny is needed for the flowchart of the computing process, and the procedures need to be reexamined, maintained, and reloaded frequently for any change – even a minor one.

In this section we will present an enterprise bonus computation system implemented in both traditional monolithic and table-driven manners. The nature of bonus calculation is quite similar to the forecasting process described at the beginning of this paper – there are lots of straightforward rules dictating how computing should be done.

3.1 A Monolithic Implementation

A common way to compute values like bonuses is to store business logic in one or a small number of complicated internal or external procedures – thus it is *monolithic*. We start by implementing a couple of tables for this application: Employees, EmployeeBonus (see table 1). In order to calculate a year-end bonus for every employee, we also define a stored function

```
CREATE OR REPLACE FUNCTION
  GetBonus(EmployeeID    IN INTEGER,
           Department    IN VARCHAR,
           BaseSalary    IN NUMBER,
           BonusName     IN VARCHAR,
           BonusArgument IN NUMBER)
  RETURN NUMBER AS
BEGIN
  IF(Department='CEO')
  THEN RETURN 2000000; END IF;
  IF(BonusName='Bonus for managers') THEN
    RETURN BaseSalary * (0.10 +
           (CASE WHEN BonusArgument >= -0.05
            THEN BonusArgument ELSE 0 END));
  END IF;
  IF(BonusName='Bonus for Sales personnel')
  THEN RETURN BonusArgument; END IF;
  IF(BonusName='Bonus for Production personnel')
  THEN RETURN BaseSalary * (0.02 +
         (CASE WHEN BonusArgument >= -0.005
          THEN BonusArgument ELSE 0 END));
  END IF;
  IF(BonusName='Bonus for R&D personnel')
  THEN RETURN 1000 * BonusArgument; END IF;
  IF(BonusName='Bonus for HR personnel')
  THEN RETURN 500; END IF;
  IF(BonusName='Personal Achievement Bonus')
  THEN RETURN 1000; END IF;
  RETURN NULL;
END GetBonus;
```

Figure 1: The GetBonus stored function.

called *GetBonus*. This function takes several arguments which can be used for different scenarios. The bonus computing process is then a query that joins Employee and EmployeeBonus and applies GetBonus on their attributes.

3.2 An ORDBMS Table-Driven Implementation

An alternative design is to follow the table-driven methodology. Besides the tables defined in section 3.1, another table named *Bonuses* (see table 3) is created to store bonus information that includes a function attribute called BonusFunction. It has a type of (NUMBER, NUMBER) → NUMBER which means that a bonus formula takes two NUMBER arguments and returns a NUMBER as the result. However, if we use object types to emulate function attributes (see section 4 and figure 3), then the business logic is wrapped in object instances. For this emulation, the base object type, *Bonus*, contains a method called *eval*. During computing, a query will choose actual arguments to join with bonus formulas. In this design, we stipulate that the first argument is the base salary of an employee (from Employees.BaseSalary) and the second is an additional factor (from EmployeeBonus.BonusArgument). Some sample data for Bonuses is listed in table 2. From the sample data, a great diversity of bonus formulas can be devised for employees in different positions and departments. The formulas can also be easily modified through DML operations. Finally, the qeury defined in figure 2 is used to compute bonuses. It is a simple aggregate that joins Employees, EmployeeBonus, and Bonuses

Table 3: Table Bonuses

Bonuses

BonusName	BonusFunction	Description

```
SELECT
 Employees.EmployeeID                    EmployeeID
,SUM(Bonuses.BonusFunction.eval(
       Employees.BaseSalary,
       EmployeeBonus.BonusArgument))  BonusSum
FROM Employees
LEFT JOIN EmployeeBonus ON
  Employees.EmployeeID = EmployeeBonus.EmployeeID
LEFT JOIN Bonuses ON
  EmployeeBonus.BonusName = Bonuses.BonusName
GROUP BY Employees.EmployeeID;
```

Figure 2: A table-driven query to compute bonuses.

and applies BonusFunction.eval on attributes.

Comparing the monolithic implementation (section 3.1) and table-driven implementation (section 3.2), it is quite obvious that the table-driven design can provide real great advantage in modeling and maintenance – data and functions are integrated in a relational manner.

4 TABLE-DRIVEN PROGRAMMING IN ORDBMS

In section 3.2, we have seen an enterprise application implemented in the table-driven style. A natural question is: how can we implement this application in an existing relational database? To support table-driven programming, a relational database must be able to store business logic – in the form of function – as an attribute of a table. As alluded in section 3.2, one answer is to use an ORDBMS that supports *dynamic method dispatch* (or *polymorphism*). Among commercial and publicly available ORDBMS systems, we found that ORACLE (ORACLE et al., 2003) and IBM DB2 (IBM et al., 2002), to our best knowledge, are the only two supporting dynamic method dispatch. The first step of implementing table-driven programming in an ORDBMS is to create an object hierarchy. Taking the object hierarchy of figure 3 as an example, a base object type *Bonus* is created with a method

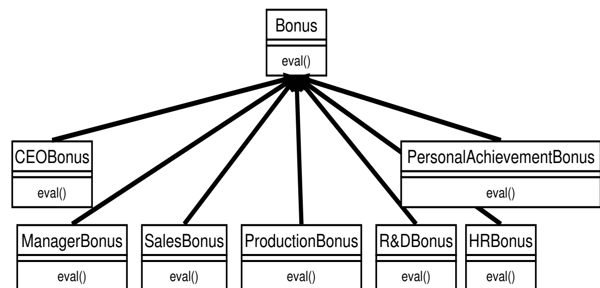

Figure 3: An object hierarchy for computing year-end bonuses for a table-driven payroll system.

Table 2: Sample data for the Bonuses table

Bonus Name	Bonus Formula ((NUMBER, NUMBER)⟶NUMBER)	Description
Bonus for the CEO	(DummyArgument0, DummyArgument1) ⟶ 2,000,000	The year-end bonus for the CEO is $2,000,000.
Bonus for managers	(BaseSalary, Performance) ⟶ BaseSalary × (0.10 + (CASE WHEN Performance ≥ -0.05 THEN Performance ELSE 0 END))	The year-end bonus for a manager is determined by a function of their Performance.
Bonus for Sales personnel	(DummyArgument0, Commission) ⟶ Commission	The year-end bonus for Sales personnel is a lump sum commission
Bonus for Production personnel	(BaseSalary, Performance) ⟶ BaseSalary × (0.02 + (CASE WHEN Performance ≥ -0.005 THEN Performance ELSE 0 END))	The year-end bonus for Production personnel is determined by a function of their Performance.
Bonus for R&D personnel	(DummyArgument0, NumberOfProjectsCompleted) ⟶ 1000 × NumberOfProjectsCompleted	The year-end bonus for R&D personnel is determined by a function of the number of completed research projects.
Bonus for HR personnel	(DummyArgument0, DummyArgument1) ⟶ 500	The year-end bonus for a HR personnel is $500.
Personal Achievement Bonus	(DummyArgument0, DummyArgument1) ⟶ 1000	The personal achievement bonus is $1,000.

called *eval*. The overriding child implementations of this method can be dynamically dispatched and used for table-driven programming.

Using ORDBMS's dynamic method dispatch is a natural solution to implement table-driven programming in SQL. However, object types were not designed specifically for table-driven programming and thus there are some limitations with this approach:

- **Object type is quite heavy**. Object types are created and maintained in data definition language (DDL) and stored in system catalog. A table-driven application may possess thousands or even millions of items of business logic. Creating millions of object types to wrap this business logic can create a great burden on the database system catalog.
- **Code wrapped in objects is still not data**. Because code wrapped in methods are attached to objects, they are not data (Gray et al., 2003) in the sense that they cannot be accessed or manipulated like ordinary data.
- **Dynamic dispatch process is complex**. To dynamically look for a pertinent method in an object type hierarchy, an ORDBMS execution engine would search either from the root object (DB2's approach) or from leaf objects (ORACLE's approach). Either way is quite complex.

5 RELATED WORK

The idea of storing functional *expressions* (not functions) as data values has been proposed before. Stonebraker's "Quel as a Data Type" (Stonebraker et al., 1984; Stonebraker et al., 1987) treated QUEL queries, statements and commands as a specialized kind of string. A stored query could be used to represent a set of records. The paper appeared just as the object-oriented database movement started to gain momentum, and perhaps for this reason failed to be implemented in commercial systems.

The work of SQL Spreadsheets (Witkowski et al., 2003) introduces spreadsheet-like computations into RDBMS through SQL extensions. It provides an efficient alternative to simplify complicated OLAP queries that otherwise are implemented in nested views, subqueries, and complex joins. However spreadsheet formulas are statically embedded in new SQL clauses, making it difficult to handle either dynamically changing formulas or a large number of business formulas.

6 CONCLUSIONS

In this paper, we discuss the idea of applying the *table-driven programming* methodology in SQL for enterprise database applications. The main goal is to integrate data and functions within a RDBMS in a relational manner. Using an existing ORDBMS, developers can use a flat object type hierarchy with polymorphic methods embedded in child types to emulate this table-driven design. However, code wrapped in an object is still not data and heavy – management of object methods relies on DDL statements, instead of DML. To avoid ORDBMS limitations, we currently work on putting *lightweight functions* in RDBMS to allow direct table-driven programming in SQL.

REFERENCES

Gray, J. et al. (2003). The Lowell Report. In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *SIGMOD 2003*, page 680. ACM.

IBM et al. (2002). *IBM DB2 Universal Database SQL Reference Volume 1 Version 8*. IBM.

ORACLE et al. (2003). *Oracle Database Application Developer's Guide - Object-Relational Features 10g Release 1 (10.1)*. Oracle.

Stonebraker, M., Anderson, E., Hanson, E. N., and Rubenstein, W. B. (1984). Quel as a Data Type. In Yormark, B., editor, *SIGMOD 1984*, pages 208–214. ACM Press.

Stonebraker, M., Anton, J., and Hanson, E. N. (1987). Extending a Database System with Procedures. *TODS*, 12(3):350–376.

Witkowski, A. et al. (2003). Spreadsheets in RDBMS for OLAP. In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *SIGMOD 2003*, pages 52–63. ACM.