

Designing Triggers with Trigger-By-Example

Dongwon Lee¹, Wenlei Mao², Henry Chiu³, Wesley W. Chu²

¹School of Information Sciences and Technology, Penn State University, PA, USA

²Department of Computer Science, University of California at Los Angeles, CA, USA

³IBM Silicon Valley Lab, CA, USA

Abstract. One of the obstacles that hinder database trigger systems from their wide deployment is the lack of tools that aid users in creating trigger rules. Similar to understanding and specifying database queries in SQL3, it is difficult to visualize the meaning of trigger rules. Furthermore, it is even more difficult to write trigger rules using such text-based trigger rule languages as SQL3. In this paper, we propose TBE (Trigger-By-Example) to remedy such problems in writing trigger rules *visually* by using QBE (Query-By-Example) ideas. TBE is a visual trigger rule composition system that helps the users understand and specify active database triggers. TBE retains benefits of QBE while extending features to support triggers. Hence, TBE is a useful tool for novice users to create simple triggers in a visual and intuitive manner. Further, since TBE is designed to hide the details of underlying trigger systems from users, it can be used as a universal trigger interface.

Keywords: Active database; Query-by-example; Triggers; Visual querying

1. Introduction

Triggers provide a facility to autonomously react to database events by evaluating a data-dependent condition and by executing a reaction whenever the condition is satisfied. Such triggers are regarded as an important database feature and are implemented by most major database vendors. Despite their diverse potential usages, one of the obstacles that hinder triggers from their wide deployment is the lack of tools that aid users in creating complex trigger rules. In many environments, the correctness of the written trigger rules is crucial since the semantics encoded in the trigger rules are shared by many applications. Although the majority of the users of triggers are DBAs or savvy end-users, writing *correct* and *complex* trigger rules is still a daunting task.

Received 6 October 2001

Revised 19 September 2002

Accepted 17 April 2003

Published online ■ ■ ■ – © Springer-Verlag ■ ■ ■

On the other hand, QBE (Query-By-Example) has been very popular since its introduction decades ago and its variants are currently being used in most modern database products. As it is based on domain relational calculus, its expressive power proves to be equivalent to that of SQL, which is based on tuple relational calculus (Codd 1972). As opposed to SQL, in which the user must conform to the phrase structure strictly, QBE users may enter any expression as an entry insofar as it is syntactically correct. That is, since the entries are bound to the table skeleton, the user can only specify admissible queries (Zloof 1977). We proposed TBE (Trigger-By-Example) (Lee et al. 2000b) as a novel graphical interface for writing triggers. Since most trigger rules are complex combinations of SQL statements, by using QBE as a user interface for triggers the user may create only admissible trigger rules. TBE uses QBE in a *declarative* fashion for writing the *procedural* trigger rules (Cochrane et al. 1996). In this paper, we discuss the design and implementation issues of TBE. Further, we present a design to make TBE a universal trigger rule formation tool that hides much of the peculiarity of the underlying trigger systems. However, it is worthwhile to point out that, in this paper, we do not address other important and arguably harder problems related to triggers (e.g., precise semantics of composite triggers, complex interaction among multiple triggers). A preliminary discussion of our work appeared in Lee et al. (2000a,b). This paper unifies and integrates these two research results.

To facilitate discussion, we shall briefly review SQL3 triggers and QBE in the following subsections.

1.1. SQL3 Triggers

Triggers play an important role in monitoring and reacting to specific changes that occur to database systems. In SQL3, *triggers*, also known as *event-condition-action rules (ECA rules)*, consist of three parts: event, condition, and action. We base our discussion on the ANSI X3H2 SQL3 working draft (Melton (ed.) 1999). The following is a definition of SQL3:

Example 1. SQL3 triggers definition.

```

<SQL3-trigger> ::= CREATE TRIGGER <trigger-name>
    {AFTER | BEFORE} <trigger-event> ON <table-name>
    [REFERENCING <references>]
    [FOR EACH {ROW | STATEMENT}]
    [WHEN <SQL-statements>]
    <SQL-procedure-statements>
<trigger-event> ::= INSERT | DELETE | UPDATE [OF <column-names>]
<reference> ::= OLD [AS] <old-value-tuple-name> |
    NEW [AS] <new-value-tuple-name> |
    OLD_TABLE [AS] <old-value-table-name> |
    NEW_TABLE [AS] <new-value-table-name>

```

1.2. QBE (Query-By-Example)

QBE is a query language as well as a visual user interface. In QBE, *programming* is done within two-dimensional skeleton tables. This is accomplished by filling in an

example of the answer in the appropriate table spaces (thus the name “by-example”). Another two-dimensional object is the *condition box*, which is used to express one or more desired conditions difficult to express in the skeleton tables. By QBE convention, variable names are lowercase alphabets prefixed with “_”, system commands are uppercase alphabets suffixed with “.”, and constants are denoted without quotes unlike SQL3. Let us see a QBE example. The following schema is used throughout the paper.

Example 2. Define the emp and dept relations with keys underlined. emp.DeptNo and dept.MgrNo are foreign keys referencing dept.Dno and emp.Eno attributes, respectively.

```
emp(Eno, Ename, DeptNo, Sal)
dept(Dno, Dname, MgrNo)
```

Example 3 shows two equivalent representations of the query in SQL3 and QBE.

Example 3. Who is being managed by the manager Tom?

```
SELECT E2.Ename
FROM emp E1, emp E2, dept D
WHERE E1.Ename = 'Tom' AND E1.Eno = D.MgrNo AND E2.DeptNo = D.Dno
```

emp	Eno	Ename	DeptNo	Sal	dept	Dno	Dname	MgrNo
	_e	Tom P.	_d			_d		_e

The rest of this paper is organized as follows. Section 2 gives a detailed description of TBE. Then, Sect. 3 illustrates a few complex TBE examples. The design and implementation of TBE, especially its translation algorithms, are discussed in Sect. 4. Section 5 presents the design of some extensions that we are planning for the TBE. Related work and concluding remarks are given in Sects. 6 and 7, respectively.

2. TBE: Trigger-By-Example

We propose to use QBE as a user interface for writing trigger rules. Our tool is called Trigger-By-Example (TBE) and has the same spirit as that of QBE. The philosophy of QBE is to require the user to know very little in order to get started and to minimize the number of concepts that he or she subsequently has to learn to understand and use the whole language (Zloof 1977). By using QBE as an interface, we attain the same benefits for creating trigger rules.

2.1. Difficulty of Expressing Procedural Triggers in Declarative QBE

Triggers in SQL3 are procedural in nature. Trigger actions can be arbitrary SQL procedural statements, allowing not only SQL data statements (i.e., select, project, join) but also transaction, connection, session statements.¹ Also, the order among action statements needs to be obeyed faithfully to preserve the correct semantics.

¹ The SQL3 triggers definition in Melton (ed.) (1999) leaves it implementation-defined whether the transaction, connection, or session statements should be contained in the action part or not.

On the contrary, QBE is a declarative query language. While writing a query, the user does not have to know if the first row in the skeleton tables must be executed before the second row or not. That is, the order is immaterial. Also QBE is specifically designed as a tool for only 1) data retrieval queries (i.e., SELECT), 2) data modification queries (i.e., INSERT, DELETE, UPDATE), and 3) schema definition and manipulation queries. Therefore, QBE cannot really handle other procedural SQL statements such as transaction or user-defined functions in a simple manner. Thus, our goal is to develop a tool that can represent the *procedural* SQL3 triggers in its entirety while retaining the *declarative* nature of QBE as much as possible.

In what follows, we shall describe how QBE was extended to be TBE, what design options were available, and which option was chosen by what rationale, etc.

2.2. TBE Model

SQL3 triggers use the ECA (Event, Condition and Action) model. Therefore, triggers are represented by three independent E, C, and A parts. In TBE, each E, C, and A part maps to the corresponding skeleton tables and condition boxes separately. To differentiate among these three parts, each skeleton table name is prefixed with its corresponding flags, E., C., or A.. The condition box in QBE is extended similarly. For instance, a trigger condition statement can be specified in the C. prefixed skeleton table and/or condition box.

C.emp	Eno	Ename	DeptNo	Sal

C.conditions

SQL3 triggers allow only INSERT, DELETE, and UPDATE as legal event types. QBE uses I., D., and U. to describe the corresponding data manipulations. TBE thus uses these constructs to describe the trigger event types. Since INSERT and DELETE always affect the whole tuple, not individual columns, I. and D. must be filled in the leftmost column of the skeleton table. Since UPDATE event can affect individual columns, U. must be filled in the corresponding columns. Otherwise, U. is filled in the leftmost column to represent that UPDATE event is monitored on all columns. Consider the following example.

Example 4. Skeleton tables (1) and (2) depict INSERT and DELETE events on the dept table, respectively. (3) depicts UPDATE event of columns Dname and MgrNo. Thus, changes occurring on other columns do not fire the trigger. (4) depicts UPDATE events of any columns on the dept table.

(1)	<table border="1"> <thead> <tr> <th>E.dept</th> <th>Dno</th> <th>Dname</th> <th>MgrNo</th> </tr> </thead> <tbody> <tr> <td>I.</td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>	E.dept	Dno	Dname	MgrNo	I.			
E.dept	Dno	Dname	MgrNo						
I.									

(2)	<table border="1"> <thead> <tr> <th>E.dept</th> <th>Dno</th> <th>Dname</th> <th>MgrNo</th> </tr> </thead> <tbody> <tr> <td>D.</td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>	E.dept	Dno	Dname	MgrNo	D.			
E.dept	Dno	Dname	MgrNo						
D.									

(3)	<table border="1"> <thead> <tr> <th>E.dept</th> <th>Dno</th> <th>Dname</th> <th>MgrNo</th> </tr> </thead> <tbody> <tr> <td> </td> <td> </td> <td>U.</td> <td>U.</td> </tr> </tbody> </table>	E.dept	Dno	Dname	MgrNo			U.	U.
E.dept	Dno	Dname	MgrNo						
		U.	U.						

(4)	<table border="1"> <thead> <tr> <th>E.dept</th> <th>Dno</th> <th>Dname</th> <th>MgrNo</th> </tr> </thead> <tbody> <tr> <td>U.</td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>	E.dept	Dno	Dname	MgrNo	U.			
E.dept	Dno	Dname	MgrNo						
U.									

Note also that since the SQL3 triggers definition requires that each trigger rule monitors only one event, there cannot be more than one row having an I., D., or U.

flag. Therefore, the same trigger action for different events (e.g., “abort when either INSERT or DELETE occurs”) needs to be expressed as separate trigger rules in SQL3 triggers.

2.3. Trigger Name

A unique name for each trigger rule needs to be set in a special input box, called the *name box*, where the user can fill in an arbitrary identifier as shown below:

<TriggerRuleName>

Typically, the user first decides the trigger name and then proceeds to the subsequent tasks. There are often cases when multiple trigger rules are written together in a single TBE query. For such cases, the user needs to provide a unique trigger name for each rule in the TBE query separately. In what follows, when there is only a single trigger rule in the example, we take the liberty of not showing the trigger name in the interest of brevity.

2.4. Triggers’ Activation Time and Granularity

The SQL3 triggers have a notion of the *event activation time* that specifies if the trigger is executed before or after its event and the *granularity* that defines how many times the trigger is executed for the particular event.

1. The activation time can have two modes, *before* and *after*. The *before* mode triggers execute before their event and are useful for conditioning the input data. The *after* mode triggers execute after their event and are typically used to embed application logic (Cochrane et al. 1996). In TBE, two corresponding constructs, `BFR.` and `AFT.`, are introduced to denote these modes. The “.” is appended to denote that these are built-in system commands.
2. The granularity of a trigger can be specified as either *for each row* or *for each statement*, referred to as *row-level* and *statement-level* triggers, respectively. The row-level triggers are executed after each modification to tuple, whereas the statement-level triggers are executed once for an event regardless of the number of the tuples affected. In TBE notation, `R.` and `S.` are used to denote the row-level and statement-level triggers, respectively.

Consider the following illustrating example.

Example 5. SQL3 and TBE representation for a trigger with *after* activation time and *row-level* granularity.

```
CREATE TRIGGER AfterRowLevelRule
AFTER UPDATE OF Ename, Sal ON emp FOR EACH ROW
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.		U.		U.

2.5. Transition Values

When an event occurs and values change, trigger rules often need to refer to the *before* and *after* values of certain attributes. These values are referred to as the *transition values*. In SQL3, these transition values can be accessed by either transition variables (i.e., OLD, NEW) or tables (i.e., OLD_TABLE, NEW_TABLE) depending on the type of triggers, whether they are row-level or statement-level. Furthermore, in SQL3, the INSERT event trigger can only use NEW or NEW_TABLE, while the DELETE event trigger can only use OLD or OLD_TABLE to access transition values. However, the UPDATE event trigger can use both transition variables and tables. We have considered the following two approaches to introduce the transition values in TBE.

1. *Using the new built-in functions*: Special built-in functions (i.e., OLD_TABLE() and NEW_TABLE() for statement-level, OLD() and NEW() for row-level) are introduced. The OLD_TABLE() and NEW_TABLE() functions return a set of tuples with values before and after the changes, respectively. Similarly the OLD() and NEW() return a single tuple with values, before and after the change, respectively. Therefore, applying aggregate functions such as CNT. or SUM. to the OLD() or NEW() is meaningless (i.e., CNT.NEW(_s) is always 1 or SUM.OLD(_s) is always same as _s). Using the new built-in functions, for instance, the event “every time more than 10 new employees are inserted” can be represented as follows:

E.emp	Eno	Ename	DeptNo	Sal
AFT.I.S.	_n			

E.conditions
CNT.ALL.NEW_TABLE(_n) > 10

Also, the event “when salary is doubled for each row” can be represented as follows:

E.emp	Eno	Ename	DeptNo	Sal	E.conditions
AFT.U.R.				_s	NEW(_s) > OLD(_s) * 2

It is not possible to apply the NEW() or NEW_TABLE() to the variable defined on the DELETE event. This is also true for the application of OLD() or OLD_TABLE() to the variable defined on the INSERT event. Asymmetrically, it is redundant to apply the NEW() or NEW_TABLE() to the variable defined on the INSERT event. Similarly, it is not possible to apply the OLD() or OLD_TABLE() to the variable defined on the DELETE event. For instance, in the above event “every time more than 10 new employees are inserted”, _n and NEW_TABLE(_n) are equivalent. Therefore, the condition expression at the condition box can be rewritten as “CNT.ALL._n > 10”. It is ambiguous, however, to simply refer to the variable defined in the UPDATE event without the built-in functions. That is, in the event “when salary is doubled for each row”, _s can refer to values both before and after the UPDATE. That is, “_s > _s * 2” at the condition box would cause an error due to its ambiguity. Therefore, for the UPDATE event case, one needs to explicitly use the built-in functions to access transition values.

2. *Using modified skeleton tables:* Depending on the event type, skeleton tables are modified accordingly; additional columns may appear in the skeleton tables.² For the INSERT event, a keyword NEW_ is prepended to the existing column names in the skeleton table to denote that these are newly inserted ones. For the DELETE event, a keyword OLD_ is prepended similarly. For the UPDATE event, a keyword OLD_ is prepended to the existing column names whose values are updated in the skeleton table to denote values before the UPDATE. At the same time, additional columns with a keyword NEW_ appear to denote values after the UPDATE. If the UPDATE event is for all columns, then OLD_ column-name and NEW_ column-name appear for all columns.

Consider an event “when John’s salary is doubled within the same department”. Here, we need to monitor two attributes – Sal and DeptNo. First, the user may type the event activation time and granularity information at the leftmost column as shown in the first table. Then, the skeleton table changes its format to accommodate the UPDATE event effect, as shown in the second table. That is, two more columns appear and the U. construct is relocated to the leftmost column.

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.			U.	U.

E.emp	Eno	Ename	OLD_DeptNo	NEW_DeptNo	OLD_Sal	NEW_Sal
AFT.U.R.						

Then, the user fills in variables into the proper columns to represent the conditions. For instance, “same department” is expressed by using same variable `_d` in both OLD_DeptNo and NEW_DeptNo columns.

E.emp	Eno	Ename	OLD_DeptNo	NEW_DeptNo	OLD_Sal	NEW_Sal
AFT.U.R.		John	<code>_d</code>	<code>_d</code>	<code>_o</code>	<code>_n</code>

E.conditions
<code>_n > _o * 2</code>

We chose the approach using new built-in functions to introduce transition values into TBE. Although there is no difference with respect to the expressive power between two approaches, the first one does not incur any modifications to the skeleton tables, thus minimizing cluttering of the user interface.

² We have also considered modifying tables, instead of columns. For instance, for the INSERT event, a keyword NEW_ is prepended to the *table* name. For the UPDATE event, a keyword OLD_ is prepended to the *table* name while new table with a NEW_ prefix is created. This approach, however, was not taken because we wanted to express column-level UPDATE event more explicitly. That is, for an event “update occurs at column Sal”, we can add only OLD_Sal and NEW_Sal attributes to the existing table if we use the “modifying columns” approach. If we take the “modifying tables” approach, however, we end up with two tables with all redundant attributes whether they are updated or not (e.g., two attributes OLD_emp.Ename and NEW_emp.Ename are unnecessarily created; one attribute emp.Ename is sufficient since no update occurs for this attribute).

2.6. The REFERENCING Construct

SQL3 allows the renaming of transition variables or tables using the REFERENCING construct for the user's convenience. In TBE, this construct is not needed since the transition values are directly referred to by the variables filled in the skeleton tables.

2.7. Procedural Statements

When arbitrary SQL procedural statements (i.e., IF, CASE, assignment statements, etc.) are written in the action part of the trigger rules, it is not straightforward to represent them in TBE due to their procedural nature. Because their expressive power is beyond what the declarative QBE, and thus TBE described so far, can achieve, we instead provide a special kind of box, called *statement box*, similar to the condition box. The user can write arbitrary SQL procedural statements delimited by “;” in the statement box. Since the statement box is only allowed for the action part of the triggers, the prefix A. is always prepended. For example,

A.statements
IF (X > 10) ROLLBACK;

2.8. The Order among Action Trigger Statements

SQL3 allows multiple action statements in triggers, each of which is executed according to the order they are written. To represent triggers whose semantics depend on the assumed sequential execution, TBE uses an implicit agreement; like Prolog, the execution order follows from top to bottom. Special care needs to be taken in translation time for such action statements as follows:

- The *action* skeleton tables appearing before are translated prior to that appearing after.
- In the same *action* skeleton tables, action statements written at the top row are translated prior to those written at the bottom one.

2.9. Expressing Conditions in TBE

In most active database triggers languages, the event part of the triggers language is exclusively concerned with what has happened and cannot perform tests on values associated with the event. Some triggers languages (e.g., Ode (Agrawal and Gehani 1989), SAMOS (Gatzia and Dittrich 1998), Chimera (Ceri et al. 1996)), however, provide filtering mechanisms that perform tests on event parameters (see Paton (ed.) (1998), Chap. 4). Event filtering mechanisms can be very useful in optimizing trigger rules; only events that passed the parameter filtering tests are sent to the condition module to avoid unnecessary expensive condition evaluations.

In general, we categorize condition definitions of the triggers into 1) *parameter filter (PF)* type and 2) *general constraint (GC)* type. SQL3 triggers definition does not have PF type; event language specifies only the event type, activation time and granularity information, and all conditions (both PF and GC types) need to be

expressed in the WHEN clause. In TBE, however, we decided to allow users to be able to differentiate PF and GC types by providing separate condition boxes (i.e., E. and C. prefixed ones) although it is not required for SQL3. This is because we wanted to support other trigger languages that have both PF and GC types in future.

1. *Parameter Filter Type*: Since this type tests the event parameters, the condition must use the transition variables or tables. Event examples such as “every time more than 10 new employees are inserted” or “when salary is doubled” in Sect. 2.5 are these types. In TBE, this type is typically represented in the E. prefixed condition box.
2. *General Constraint Type*: This type expresses general conditions regardless of the event type. In TBE, this type is typically represented in the C. prefixed condition boxes. One such example is illustrated in Example 6.

Example 6. When an employee’s salary is increased more than twice within the same year (a variable CURRENT_YEAR contains the current year value), log changes into the log(Eno, Sal) table. Assume that there is another table sal-change(Eno, Cnt, Year) to keep track of the employee’s salary changes.

```
CREATE TRIGGER TwiceSalaryRule AFTER UPDATE OF Sal ON emp
FOR EACH ROW
WHEN EXISTS (SELECT * FROM sal-change WHERE Eno = NEW.Eno
            AND Year = CURRENT_YEAR AND Cnt >= 2)
BEGIN ATOMIC
    UPDATE sal-change SET Cnt = Cnt + 1
        WHERE Eno = NEW.Eno AND Year = CURRENT_YEAR;
    INSERT INTO log VALUES(NEW.Eno, NEW.Sal);
END
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.	_n			U._s

C.sal-change	Eno	Cnt	Year
	NEW(_n)	_c	CURRENT_YEAR

C.conditions
_c >= 2

A.sal-change	Eno	Cnt	Year
U.	NEW(_n)	_c + 1 _c	CURRENT_YEAR

A.log	Eno	Sal
l.	NEW(_n)	NEW(_s)

Here, the condition part of the trigger rule (i.e., WHEN clause) checks the Cnt value of the sal-change table to check how many times salary was increased in the same year, and thus, does not involve testing any transition values. Therefore, it makes more sense to represent such a condition as GC type, not PF type. Note that the headers of the sal-change and condition box have the C. prefixes.

3. TBE Examples

3.1. Integrity Constraint Triggers

A trigger rule to maintain the foreign key constraint is shown below.

Example 7. When a manager is deleted, all employees in his or her department are deleted too.

```
CREATE TRIGGER ManagerDelRule AFTER DELETE ON emp
FOR EACH ROW
DELETE FROM emp E1 WHERE E1.DeptNo =
(SELECT D.Dno FROM dept D WHERE D.MgrNo = OLD.Eno)
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.D.R.	_e			

A.dept	Dno	Dname	MgrNo
	_d		_e

A.emp	Eno	Ename	DeptNo	Sal
D.			_d	

In this example, the WHEN clause is deliberately missing; that is, the trigger rule does not check if the deleted employee is in fact a manager or not because the rule deletes only the employee whose manager is just deleted. Note how _e variable is used to join the emp and dept tables to find the department whose manager is just deleted. The same query could have been written with a condition test in a more explicit manner as follows:

E.emp	Eno	Ename	DeptNo	Sal
AFT.D.R.	_e			

C.dept	Dno	Dname	MgrNo
	_d		_m

C.conditions
OLD(_e) = _m

A.emp	Eno	Ename	DeptNo	Sal
D.			_d	

Another example is shown below.

Example 8. When employees are inserted into emp table, abort the transaction if there is one violating the foreign key constraint.

```
CREATE TRIGGER AbortEmp AFTER INSERT ON emp
FOR EACH STATEMENT
WHEN EXISTS (SELECT * FROM NEW_TABLE E WHERE NOT EXISTS
(SELECT * FROM dept D WHERE D.Dno = E.DeptNo))
ROLLBACK
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.I.S.			_d	

C.dept	Dno	Dname	MgrNo
¬	_d		

A.statements
ROLLBACK

In this example, if the granularity were R. instead of S., then the same TBE query would represent different SQL3 triggers. That is, row-level triggers generated from the same TBE representation would have been:

```
CREATE TRIGGER AbortEmp AFTER INSERT ON emp
FOR EACH ROW
WHEN NOT EXISTS
    (SELECT * FROM dept D WHERE D.Dno = NEW.DeptNo)
ROLLBACK
```

We believe that this is a good example illustrating why TBE is useful in writing trigger rules. That is, when the only difference between two rules is the trigger granularity, a simple change between R. and S. is sufficient in TBE. However, in SQL3, users should devise quite different rule syntaxes as demonstrated above.

3.2. View Maintenance Triggers

Suppose a company maintains the following view derived from the emp and dept schema.

Example 9. Create a view HighPaidDept that has at least one “rich” employee earning more than 100K.

```
CREATE VIEW HighPaidDept AS
    SELECT DISTINCT D.Dname
    FROM emp E, dept D
    WHERE E.DeptNo = D.Dno AND E.Sal > 100K
```

The straightforward way to maintain the views upon changes to the base tables is to re-compute all views from scratch. Although incrementally maintaining the view is more efficient than this method, for the sake of trigger example, let us implement the naive scheme below. The following is only for an UPDATE event case.

Example 10. Refresh the HighPaidDept when UPDATE occurs on emp table.

```
CREATE TRIGGER RefreshView AFTER UPDATE OF DeptNo, Sal ON emp
FOR EACH STATEMENT
BEGIN ATOMIC
    DELETE FROM HighPaidDept;
    INSERT INTO HighPaidDept
        (SELECT DISTINCT D.Dname FROM emp E, dept D
         WHERE E.DeptNo = D.Dno AND E.Sal > 100K);
END
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.S.			U.	U.

A.emp	Eno	Ename	DeptNo	Sal
			_d	> 100K

A.dept	Dno	Dname	MgrNo
	_d	_n	

A.HighPaidDept	Dname
D.	
I.	_n

By the implicit ordering of TBE, the DELETE statement executes prior to the INSERT statement.

3.3. Replication Maintenance Triggers

Now let us consider the problem of maintaining replicated copies in synchronization with the original copy. Suppose that all changes are made to the `primary copy` while the `secondary copy` is asynchronously updated by triggering rules. Actual changes to the `primary copy` are recorded in Delta tables. Then, deltas are applied to the `secondary copy`. This logic is implemented by five trigger rules below. The first three rules monitor the base table for INSERT, DELETE, UPDATE events, respectively, and the last two rules implement the actual synchronization.

Example 11. Maintain the replicated copy `dept_copy` when the original `dept` table changes.

```
Rule 1: CREATE TRIGGER CaptureInsertRule
        AFTER INSERT ON dept FOR EACH STATEMENT
        INSERT INTO PosDelta (SELECT * FROM NEW_TABLE)
```

```
Rule 2: CREATE TRIGGER CaptureDeleteRule
        AFTER DELETE ON dept FOR EACH STATEMENT
        INSERT INTO NegDelta (SELECT * FROM OLD_TABLE)
```

```
Rule 3: CREATE TRIGGER CaptureUpdateRule
        AFTER UPDATE ON dept FOR EACH STATEMENT
        BEGIN ATOMIC
          INSERT INTO PosDelta (SELECT * FROM NEW_TABLE);
          INSERT INTO NegDelta (SELECT * FROM OLD_TABLE);
        END
```

```
Rule 4: CREATE TRIGGER PosSyncRule
        AFTER INSERT ON PosDelta FOR EACH STATEMENT
        INSERT INTO dept_copy (SELECT * FROM PosDelta)
```

```
Rule 5: CREATE TRIGGER NegSyncRule
        AFTER INSERT ON NegDelta FOR EACH STATEMENT
        DELETE FROM dept_copy WHERE Dno IN
        (SELECT Dno FROM NegDelta)
```

E.dept	Dno	Dname	MgrNo
AFT.I.S.	_i1	_i2	_i3
AFT.D.S.	_d1	_d2	_d3
AFT.U.S.	_u1	_u2	_u3

A.PosDelta	Dno	Dname	MgrNo
i.	_i1	_i2	_i3
i.	NEW_TABLE(_u1)	NEW_TABLE(_u2)	NEW_TABLE(_u3)

A.NegDelta	Dno	Dname	MgrNo
I.	_d1	_d2	_d3
I.	OLD_TABLE(_u1)	OLD_TABLE(_u2)	OLD_TABLE(_u3)

E.PosDelta	Dno	Dname	MgrNo
AFT.I.S.	_p1	_p2	_p3

E.NegDelta	Dno	Dname	MgrNo
AFT.I.S.	_n1		

A.dept_copy	Dno	Dname	MgrNo
I.	_p1	_p2	_p3
D.	_n1		

Note how multiple trigger rules (i.e., 5 rules) can be written in a unified TBE representation. This feature is particularly useful to represent multiple yet “related” trigger rules. The usage of the distinct variables for different trigger rules (e.g., `_i1`, `_d1`, `_u1`) enables the user to distinguish different trigger rules in rule generation time. However, it is worthwhile to point out that TBE does not currently support ordering among multiple trigger rules.

4. Translation Algorithm

A preliminary version of TBE prototype is implemented in Java using jdk 1.2.1 and swing 1.1 as shown in Fig. 1. More discussion about implementation-related issues can be found in Lee et al. (2000a).

Our algorithm is an extension of the algorithm by McLeod (1976), which translates from QBE to SQL. Its input is a list of skeleton tables and the condition boxes, while its output is a SQL query string. Let us denote the McLeod’s algorithm as `qbe2sql(<input>)` and ours as `tbe2triggers`.

4.1. The qbe2sql Algorithm

We have implemented basic features of the `qbe2sql` algorithm in McLeod (1976), in the exception of queries having the GROUP-BY construct. The algorithm first determines the type of query statement. The basic cases involve operators, such as SELECT, UPDATE, INSERT, and DELETE. Special cases use UNION, EXCEPT, and INTERSECT where the statements are processed recursively. General steps of the translation implemented in TBE are as follows:

1. Duplicate tables are renamed. (e.g., “FROM supply, supply” is converted into “FROM supply S1, supply S2”)
2. SELECT clause (or other type) is printed by searching through TBETables’ fields for projection (i.e., `P.` command). Then, FROM clause is printed from TBETable table names.
3. Example variables are extracted from TBETables by searching for tokens starting with “_”. Variables with the same names indicate table joins; table names and corresponding column names of the variables are stored.

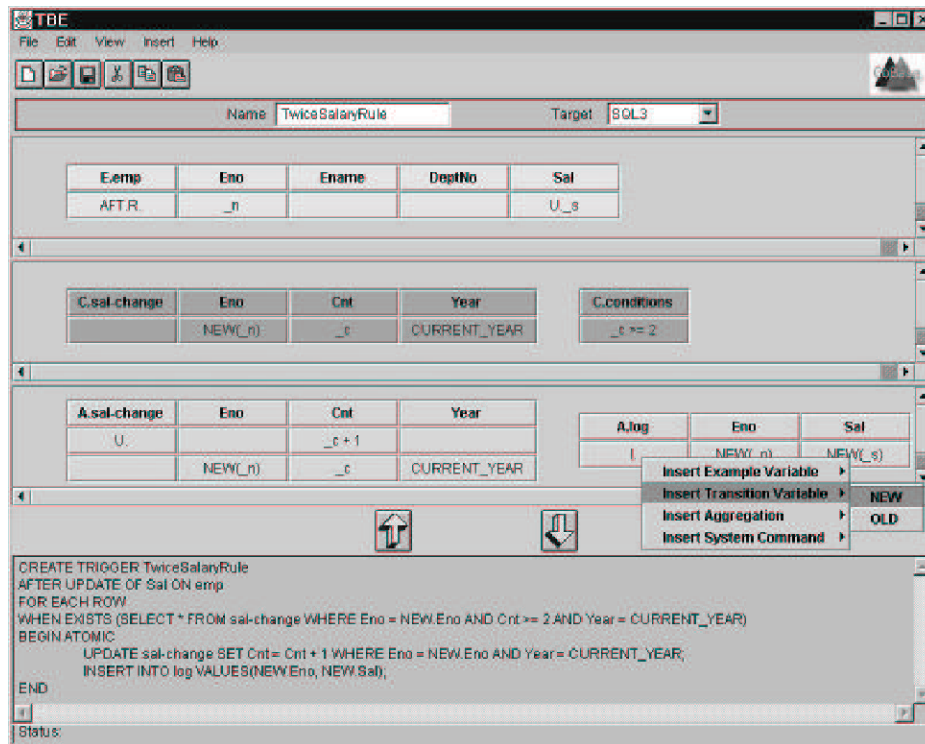


Fig. 1. The screen dump of the TBE prototype.

4. Process conditions. Variables are matched with previously extracted variables and replaced with corresponding table and column names. (e.g., a variable `_n` at column `Eno` of the table `emp` is replaced to `emp.Eno`). Constants are handled accordingly as well.

4.2. The `tbe2triggers` Algorithm

Let us assume that `_var` is an example variable filled in some column of the skeleton table. `colname(_var)` is a function to return the column name given the variable name `_var`. Skeleton tables and condition or statement boxes are collectively referred to as *entries*.

1. *Preprocessing*: This step does two tasks: 1) reducing the TBE query to an equivalent, but simpler form by moving the condition box entries to the skeleton tables, and 2) partitioning the TBE query into distinct groups when multiple trigger rules are written together. This can be done by comparing variables filled in the skeleton tables and collecting those entries with the same variables being used in the same group. Then, apply the following steps 2, 3, and 4 to each distinct group repeatedly to generate separate trigger rules.
2. *Build event clause*: Input all the `E.` prefixed entries. The “CREATE TRIGGER <trigger-name>” clause is generated by the trigger name <trigger-name> filled in the name box. By checking the constructs (e.g., `AFT.`, `R.`), the system can

determine the activation time and granularity of the triggers. The event type can also be detected by constructs (e.g., I., D., U.). If U. is found in the individual columns, then the “AFTER UPDATE OF <column-names>” clause is generated by enumerating all column names in an arbitrary order. Then,

- (a) Convert all variables $_var_i$ used with I. event into NEW($_var_i$) (if row-level) or NEW_TABLE($_var_i$) (if statement-level) accordingly.
 - (b) Convert all variables $_var_i$ used with D. event into OLD($_var_i$) (if row-level) or OLD_TABLE($_var_i$) (if statement-level) accordingly.
 - (c) If there is a condition box or a column having comparison operators (e.g., <, \geq) or aggregation operators (e.g., AVG., SUM.), gather all the related entries and pass them over to step 3.
3. *Build condition clause:* Input all the C. prefixed entries as well as the E. prefixed entries passed from the previous step.
- (a) Convert all built-in functions for transition values and aggregate operators into SQL3 format. For instance, OLD($_var$) and SUM. $_var$ are converted into OLD.name and SUM(name) respectively, where name = colname($_var$).
 - (b) Fill P. command in the table name column (i.e., leftmost one) of all the C. prefixed entries unless they already contain P. commands. This will result in creating “SELECT $table_1.*$, . . . , $table_n.*$ FROM $table_1$, . . . , $table_n$ ” clause.
 - (c) Gather all entries into the <input> list and invoke the qbe2sql(<input>) algorithm. Let the returned SQL string as <condition-statement>. For row-level triggers, create a “WHEN EXISTS (<condition-statement>)” clause. For statement-level triggers, create “WHEN EXISTS (SELECT * FROM NEW_TABLE (or OLD_TABLE) WHERE (<condition-statement>))”
4. *Build action clause:* Input all the A. prefixed entries.
- (a) Convert all built-in functions for transition values and aggregate operators into the SQL3 format like in step 3.(a).
 - (b) Partition the entries into distinct groups. That is, gather entries with identical variables being used in the same group. Each group will have one data modification statement such as INSERT, DELETE, or UPDATE. Preserve the order among partitioned groups.
 - (c) For each group G_i , invoke the qbe2sql(< G_i >) algorithm according to the order in step 4.(b). Let the resulting SQL string for G_i be <action-statement> $_i$. The contents in the statement box are literally copied to <action-statement> $_i$. Then, final action statements for triggers would be “BEGIN ATOMIC <action-statement> $_1$; . . . , <action-statement> $_n$; END”.

5. TBE as a Universal Trigger Rule Formation Tool

At present, TBE supports only SQL3 triggers syntax. Although SQL3 is close to its final form, many database vendors are already shipping their products with their own proprietary triggers syntax. When multiple databases are interconnected or integrating one database to another, these diversities can introduce significant problems. To remedy this problem, one can use TBE as a universal triggers construction tool. The user can create trigger rules using the TBE interface and save them as TBE’s internal format. When there is a need to change one database to another, the user can reset the target system (e.g., from Oracle to DB2) to re-generate new trigger rules.

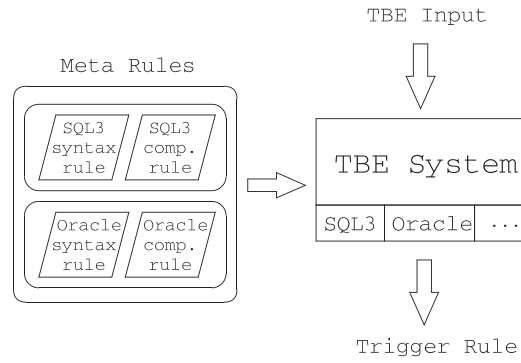


Fig. 2. The architecture of TBE as a universal triggers construction tool.

Ideally, we would like to be able to add new types of database triggers in a *declarative* fashion. That is, given a new triggers system, a user needs only to describe what kind of syntax the triggers use. Then, TBE should be able to generate the target trigger rules without further intervention from the user. Two inputs to TBE are needed to add new database triggers: the *trigger syntax rule* and *trigger composition rule*. In a trigger syntax rule, a detailed description of the syntactic aspect of the triggers is encoded by the declarative language. In a trigger composition rule, information as to how to compose the trigger rule (i.e., English sentence) using the trigger syntax rule is specified. The behaviour and output of TBE conforms to the specifics defined in the meta rules of the selected target trigger system. When a user chooses the target trigger system in the interface, corresponding trigger syntax and composition rules are loaded from the meta rule database into the TBE system. The high-level overview is shown in Fig. 2.

5.1. Trigger Syntax Rule

TBE provides a declarative language to describe trigger syntax, whose EBNF is shown below:

```

<Trigger-Syntax-Rule> ::= <event-rule> | <condition-rule> | <action-rule>
<event-rule> ::= 'event' 'has' <event-rule-entry> (',' <event-rule-entry>)* ';'
<event-rule-entry> ::= <structure-operation> 'on' ('row' | 'attribute') |
    <activation-time> | <granularity> | <evaluation-time>
<structure-operation> ::= ('I.' | 'D.' | 'U.' | 'RT.') 'as' <value>
<activation-time> ::= ('BFR.' | 'AFT.' | 'ISTD.') 'as' <value>
<granularity> ::= ('R.' | 'S.') 'as' <value>
<value> ::= <identifier> | ' ' <identifier> ' ' | 'null' | 'true'
<condition-rule> ::= 'condition' 'has' <condition-rule-entry> (',' <condition-rule-entry>)* ';'
<condition-rule-entry> ::= <condition-role> | <condition-context>
<condition-role> ::= 'role' 'as' ('mandatory' | 'optional')
<condition-context> ::= 'context' 'as'
'(' ('NEW' | 'OLD' | 'NEW_TABLE' | 'OLD_TABLE') 'as' <value> ')'
<action-rule> ::= 'action' 'has' <action-rule-entry> (',' <action-rule-entry>)* ';'
<action-rule-entry> ::= <structure-operation> | <evaluation-time>
<evaluation-time> ::= ('DFR.' | 'IMM.' | 'DTC.') 'as' <value>

```


Although the detailed discussion of the language constructs is beyond the scope of this paper, the essence of the language has the form “command as value”, meaning the trigger feature *command* is supported and represented by the keyword *value*. For instance, a clause `NEW_TABLE as INSERTED` for Starburst system would mean that “Starburst supports statement-level triggering and uses the keyword `INSERTED` to access transition values”.

Example 12. SQL3 trigger syntax can be described as follows:

```
event has (
  I. as INSERT on row, D. as DELETE on row,
  U. as UPDATE on attribute,
  BFR. as BEFORE, AFT. as AFTER, R. as ROW, S. as STATEMENT
) ;
condition has (
  role as optional,
  transition as (NEW as NEW, OLD as OLD,
    NEW_TABLE as NEW_TABLE, OLD_TABLE as OLD_TABLE)
) ;
action has (
  I. as INSERT, D. as DELETE, U. as UPDATE
) ;
```

The interpretation of this meta rule should be self-describing. For instance, the fact that there is no clause `S. as ...` implies that SQL3 triggers do not support event monitoring on the selection operation. In addition, the clause `T. as STATEMENT` implies that SQL3 triggers support table-level event monitoring using the keyword “FOR EACH STATEMENT”.

The partial comparison of the trigger syntax of SQL3, Starburst, Postgres, Oracle and DB2 system is shown in Table 1. The leftmost column contains TBE commands while other columns contain equivalent keywords of the corresponding trigger system. “N/A” means the feature is not supported and “true” means the feature is supported by default. Using the language constructs defined above, these syntax can be easily encoded into the trigger syntax rule. Note that our language is limited to triggers based on the ECA and the relational data model.

5.2. Trigger Composition Rule

After the syntax is encoded, TBE still needs information on how to compose English sentences for trigger rules. This logic is specified in the trigger composition rule. In a trigger composition rule, a macro variable is surrounded by the \$ sign and substituted with actual values during rule generation time.

Example 13. The following is a SQL3 trigger composition rule:

```
CREATE TRIGGER $trigger-name$
  $activation-time$ $structure-operation$ ON $table$
  FOR EACH $granularity$
  WHEN $condition-statement$
  BEGIN ATOMIC
    $action-statement$
  END
```

Table 1. Syntax comparison of five triggers using the *trigger syntax rule*. The leftmost column contains TBE commands while other columns contain equivalent keywords of the corresponding trigger system. “N/A” means the feature is not supported and “true” means the feature is supported by default.

TBE	SQL3	Starburst	Postgres	Oracle	DB2
I. D. U. RT.	INSERT DELETE UPDATE N/A	INSERTED DELETED UPDATED N/A	INSERT DELETE UPDATE RETRIEVE	INSERT DELETE UPDATE N/A	INSERT DELETE UPDATE N/A
BFR. AFT. ISTD.	BEFORE AFTER N/A	N/A true N/A	N/A true INSTEAD	BEFORE AFTER N/A	BEFORE AFTER N/A
R. S.	ROW STATEMENT	N/A true	TUPLE N/A	ROW true	ROW STATEMENT
NEW OLD NEW_TABLE OLD_TABLE	NEW OLD NEW_TABLE OLD_TABLE	N/A N/A INSERTED, NEW-UPDATED DELETED, OLD-UPDATED	NEW CURRENT N/A N/A	NEW OLD N/A N/A	NEW OLD NEW_TABLE OLD_TABLE

In rule generation time, for instance, variable \$activation-time\$ is replaced with the value either BEFORE or AFTER since those two are the only valid values according to the trigger syntax rule in Example 12. In addition, variables \$condition-statement\$ and \$action-statement\$ are replaced with statements generated by the translation algorithm in Sect. 4.

6. Related Work

Past active database research has focused on active database rule languages (Agrawal and Gehani 1989), rule execution semantics (Cochrane et al. 1996), or rule management and system architecture issues (Simon and Kotz-Dittrich 1995). In addition, research on visual querying has been done in traditional database research (Embley 1989; Zloof 1977; Benzi et al. 1999). To a greater or lesser extent, all of this research focused on devising novel visual querying schemes to replace the data retrieval aspects of SQL language. Although some have considered data definition aspects (Collet and Brunel 1992) or manipulation aspects, none have extensively considered the *trigger* aspects of SQL, especially from the user interface point of view.

Other work, such as *IFO*₂ (Teisseire et al. 1994) or IDEA (Ceri et al. 1996), have attempted to build graphical triggers description tools, too. Using *IFO*₂, one can describe how different objects interact through events, thus giving priority to an overview of the system. Argonaut from the IDEA project (Ceri et al. 1996) focused on the automatic generation of active rules that correct integrity violation based on declarative integrity constraint specification and active rules that incrementally maintain materialized views based on view definition. TBE, on the other hand, tries to help users *directly* design active rules with minimal learning.

Other than QBE skeleton tables, *forms* have been popular building blocks for visual querying mechanism as well (Yao et al. 1984; Embley 1989). For instance, Embley (1989) proposes the NFQL as a communication language between humans and database systems. It uses forms in a strictly nonprocedural manner to represent query. Other work using forms focused on the querying aspect of the visual interface

(Collet and Brunel 1992). To the best of our knowledge, the only work that is directly comparable to ours is RBE (Chang and Chen 1997). TBE is different from RBE in the following aspects:

- Since TBE is designed with SQL3 triggers in mind, it is capable of creating all the complex SQL3 trigger rules. Since RBE's capability is limited to OPS5-style production rules, it cannot express the subtle difference of the trigger activation time or granularity.
- Since RBE focuses on building an active database system in which RBE is only a small part, no evident suggestion of QBE as a user interface to trigger construction is given. On the contrary, TBE is specifically aimed for that purpose.
- The implementation of RBE is tightly coupled with the underlying rule system and database so that it cannot easily support multiple heterogeneous database triggers. Since TBE implementation is a thin layer utilizing a translation from a visual representation to the underlying triggers, it is loosely coupled with the database.

7. Conclusion

In this paper, we presented the design and implementation of TBE, a visual trigger rule specification interface. QBE was extended to handle features specific to ECA trigger rules. TBE extends the visual querying mechanism from QBE and applies it to triggers construction applications. Examples were given to demonstrate SQL3-based trigger rule generation procedures as well as the TBE to SQL3 trigger translation algorithm. Extension of TBE toward universal trigger rule interface was also included. For a trigger system T , we can declaratively specify the syntax mapping between TBE and T , so that TBE can be used not only as a trigger rule formation tool, but also as a universal intermediary translations between supported systems.

Acknowledgements. We thank anonymous reviewers for their very useful comments and suggestions. Part of this work was done while Dongwon Lee and Henry Chiu were at UCLA.

References

- Agrawal R, Gehani N (1989) Ode (Object Database and Environment): The Language and the Data Model. ACM SIGMOD. Portland, OR
- Benzi F, Maio D, Rizzi S (1999) VISIONARY: a Viewpoint-Based Visual Language for Querying Relational Databases. *J Vis Lang Comput* 10(2):117–145
- Ceri S, Fraternali P, Paraboschi S, Tanca L (1996) Active Rule Management in Chimera. In *Active Database Systems: Triggers and Rules for Active Database Processing*, Morgan Kaufmann, pp 151–176
- Chang Y-I, Chen F-L (1997) RBE: A Rule-by-example Action Database System. *Softw Pract Exp* 27:365–394
- Cochrane R, Pirahesh H, Mattos N (1996) Integrating Triggers and Declarative Constraints in SQL Database Systems. VLDB. Mumbai (Bombay), India
- Codd EF (1972) Relational Completeness Of Data Base Languages. *Data Base Systems, Courant Computer Symposia Serie* 6:65–98
- Collet C, Brunel E (1992) Definition and Manipulation of Forms with FO2. IFIP Working Conf on Vis Database Syst (VDB)
- Embley DW (1989) NFQL: The Natural Forms Query Language. *ACM Trans Database Syst* 14(2):168–211
- Gatzju S, Dittrich KR (1998) SAMOS. In *Active Rules In Database Syst*, Springer, Berlin Heidelberg New York, pp 233–248
- Lee D, Mao W, Chiu H, Chu WW (2000a) TBE: A Graphical Interface for Writing Trigger Rules in Active Databases. IFIP Working Conf on Vis Database Syst (VDB). Fukuoka, Japan
- Lee D, Mao W, Chu WW (2000b) TBE: Trigger-By-Example. *Int Conf on Conceptual Modeling (ER)*. Salt Lake City, UT

- McLeod D (1976) The Translation and Compatibility of SEQUEL and Query by Example. Int Conf on Softw Eng (ICSE). San Francisco, CA
- Melton J (ed) (1999) ANSI/ISO Working Draft) Foundation (SQL/Foundation). Tech rep, ANSI X3H2-99-079/WG3:YGI-011, <ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-foundation-wd-1999-03.pdf>
- Paton NW (ed) (1998) Active Rules in Database Systems. Springer, Berlin Heidelberg New York
- Simon E, Kotz-Dittrich A (1995) Promises and Realities of Active Database Systems. VLDB. Zurich, Switzerland
- Teisseire M, Poncelet P, Cichetti R (1994) Towards Event-Driven Modelling for Database Design. VLDB. Santiago de Chile, Chile
- Yao SB, Hevner AR, Shi Z, Luo D (1984) FORMANAGER: An Office Forms Management System. ACM Trans Inf Syst 2(3):235–262
- Zloof MM (1977) Query-by-Example: a data base language. IBM Syst J 16(4):342–343

A. Appendix

McLeod (1976) has a QBE to SQL translation algorithm in notations somewhat different (and obsolete) from what most current DB textbooks use. In this section, we clear up those confusions and re-write all example queries in the paper in a familiar notation. This is necessary since our `tbe2triggers` algorithm is based on the McLeod's `qbe2sql` algorithm. First, examples are based on following schema:

```
emp(ENAME, SAL, MGR, DEPT)
sales(DEPT, ITEM)
supply(ITEM, SUPPLIER)
type(ITEM, COLOUR, SIZE)
```

In what follows, both the recommended QBE and SQL representations of the given query are presented. Note that there could be many other representations equivalent to what is presented here. We only showed here what we believe to be the most reasonable ones.

A.1. Simple Queries

In this section, basic QBE queries and their SQL translation are introduced. The first `qbe2sql` implementation needs to be able to handle at least all the simple queries in this section.

Query 1: Print the red items.

type	Item	Colour
	P.	red

```
SELECT Item
FROM type
WHERE Colour = 'red'
```

Query 2: Find the departments that sell items supplied by parker.

sales	Dept	Item	supply	Item	Supplier
	P.	_i		_i	parker

```
SELECT S.Dept
FROM sales S, supply T
WHERE S.item = T.item AND T.supplier = 'parker'
```

Query 3: Find the names of employees who earn more than their manager.

emp	Name	Sal	Mgr
	P. _m	_e1 _e2	_m

conditions
_e1 > _e2

```
SELECT E1.Name
FROM emp E1, emp E2
WHERE E1.Mgr = E2.Name AND E1.Sal > E2.Sal
```

Query 4: Find the departments that sell pens and pencils.

sales	Dept	Item
	P_d _d	pen pencil

```
SELECT S1.Dept
FROM sales S1, sales S2
WHERE S1.Dept = S2.Dept AND S1.Item = 'pen' AND S2.Item = 'pencil'
```

In QBE, the same query can be expressed using a condition box as follows.

sales	Dept	Item
	P.	_i

conditions
_i = (pen AND pencil)

Note that this query should not be translated into the following SQL:

```
SELECT Dept
FROM sales
WHERE Item = 'pen' AND Item = 'pencil'
```

Instead, the following SQL using INTERSECT is the correct translation.

```
(SELECT Dept FROM sales WHERE Item = 'pen')
INTERSECT
(SELECT Dept FROM sales WHERE Item = 'pencil')
```

Query 5: Find the departments that sell pens or pencils.

sales	Dept	Item
	P_d1 P_d2	pen pencil

```
(SELECT Dept FROM sales WHERE Item = 'pen')
UNION
(SELECT Dept FROM sales WHERE Item = 'pencil')
```

In QBE, the same query can be expressed using a condition box as follows.

Query 6: Same query as Query 5.

sales	Dept	Item
	P.	_i

conditions
_i = (pen OR pencil)

```
SELECT Dept
FROM sales
WHERE Item = 'pen' OR Item = 'pencil'
```

Query 7: Print all the department and supplier pairs such that the department sells an item that the supplier supplies.

sales	Dept	Item	supply	Item	Supplier
	P_d	_i		_i	P_s

```
SELECT S.Dept, T.Supplier
FROM sales S, supply T
WHERE S.Item = T.Item
```

Query 8: List all the items except the ones which come in green.

type	Item	Colour
	P_i	¬ green

```
SELECT Item
FROM type
WHERE Colour <> 'green'
```

All following QBE and SQL expressions are equivalent.

type	Item	Colour	type	Item	Colour
¬	P_i	green		P_i	green
	_i			¬_i	

```
(SELECT Item FROM type)
EXCEPT
(SELECT Item FROM type WHERE Colour = 'green')
```

```
SELECT Item FROM type WHERE Item NOT IN
(SELECT Item FROM type WHERE Colour = 'green')
```

Query 9: Find the departments that sell items supplied by parker and bic.

sales	Dept	Item	supply	Item	Supplier
	P_d	_i1		_i1	parker
	_d	_i2		_i2	bic

```
SELECT S1.Dept
FROM sales S1, sales S2, supply T1, supply T2
WHERE S1.Dept = S2.Dept AND S1.Item = T1.Item AND S2.Item = T2.Item
AND T1.Supplier = 'parker' AND T2.supplier = 'bic'
```

This could have been written using [] notation (i.e., set) as follows:

sales	Dept	Item	supply	Item	Supplier
	P_d	[_i1,_i2]		[_i1,_i2]	[parker,bic]

Query 10: Find the departments that sell items each of which is supplied by parker and bic.

sales	Dept	Item	supply	Item	Supplier
	P_d	_i		_i	[parker,bic]

```
SELECT S.Dept
FROM sales S, supply T1, supply T2
WHERE S.Item = T1.Item AND S.Item = T2.Item
AND T1.Supplier = 'parker' AND T2.supplier = 'bic'
```

A.2. Grouping Queries

In this section, more complex QBE queries and their SQL translations are introduced using grouping and aggregation on the groups. Queries are ordered according to their complexities.

Query 11: Count employees by departments and manager.

emp	Name	Dept	Mgr
	P.CNT.ALL._n	P.G.	P.G.

```
SELECT Dept, Mgr, COUNT(Name)
FROM emp
GROUP BY Dept, Mgr
```

In QBE, aggregate operators (i.e., `CNT.`, `SUM.`, `AVG.`, `MIN.`, `MAX.`) can only be applied to “set”. Hence, `CNT.ALL._n` is used instead of `CNT._n`, where `ALL.` ensures returning a *set* of employee names. In addition, in QBE, duplicates are automatically eliminated unless stated otherwise. Since the query asks the total number of all the employees regardless of their names being identical, we add `ALL.` to ensure not to eliminate duplicates.

Query 12: Among all departments with total salaries greater than 22,000, find those which sell pens.

emp	Sal	Dept	sales	Dept	Item	conditions
	_s	P.G._d		_d	pen	SUM.ALL._s > 22000

```
SELECT E.Dept
FROM emp E, sales S
WHERE E.Dept = S.Dept AND S.Item = 'pen'
GROUP BY E.Dept
HAVING SUM(E.Sal) > 22000
```

Query 13: List the name and department of each employee such that his department sells less than three items.

emp	Name	Dept	sales	Dept	Item	conditions
	P.	P._d		G._d	_i	CNT.UNQ.ALL._i < 3

```
SELECT E.Dept, E.Name
FROM emp E, sales S
WHERE E.Dept = S.Dept
GROUP BY S.Dept
HAVING COUNT(DISTINCT S.Item) < 3
```

To count the *distinct* names of the department, since QBE automatically eliminates duplicates, `CNT._i` should be enough. However, `CNT.` operator can only be applied to a set, we need to append `UNQ.ALL.` after `CNT.` operator.

Query 14: Find the departments that sell all the items of all the suppliers.

We need to check two conditions: 1) the item being sold by the department is actually supplied by some supplier, and 2) the total number of items being sold by the department is the same as the total number of items of all the suppliers.

sales	Dept	Item	supply	Item	Supplier	conditions
	P.G._d	_i1		_i1		CNT.UNQ.ALL._i1 = CNT.UNQ.ALL._i2
				_i2		

```

SELECT S.Dept
FROM sales S, supply T
WHERE S.Item = T.Item
GROUP BY S.Dept
HAVING COUNT(DISTINCT S.Item) =
      (SELECT COUNT(DISTINCT Item) FROM supply)

```

Query 15: Find the departments that sell all the items supplied by parker (and possibly some more).

sales	Dept	Item	supply	Item	Supplier
	P.G.	[ALL._i,*]		ALL._i	parker

This query first finds all the items supplied by parker. `ALL.` ensures that duplicates are kept (i.e., multi-set). Then, for each dept (i.e., G.), find department who has items that contain all the items supplied by the parker (i.e., `_i`) and some more (i.e., `*`). We can translate this into two different SQL expressions as follows: 1. *When CONTAINS operator is supported:*

```

SELECT Dept
FROM sales
GROUP BY Dept
HAVING Item CONTAINS
      (SELECT Item FROM supply WHERE Supplier = 'parker')

```

2. *When CONTAINS operator is not supported:* use the equivalence that “A contains B” is same as “not exists (B except A)”. Since CONTAINS operator is not part of the standard SQL and supported by only a few vendors, this case should be a default.

```

SELECT Dept
FROM sales
GROUP BY Dept
HAVING NOT EXISTS
      ((SELECT Item FROM supply WHERE Supplier = 'parker') EXCEPT (Item))

```

Query 16: Find the departments that sell all the items supplied by parker (and nothing more).

sales	Dept	Item	supply	Item	Supplier
	P.G.	ALL._i		ALL._i	parker

Here, we need to express *set equality* situation. To express “ $A = B$ ”, we can use “ $A - B = \emptyset$ and $B - A = \emptyset$ ”.

```

SELECT Dept
FROM sales
GROUP BY Dept
HAVING COUNT((SELECT Item FROM supply WHERE Supplier = 'parker')
      EXCEPT (Item)) = 0
      AND COUNT((Item) EXCEPT
      (SELECT Item FROM supply WHERE Supplier = 'parker')) = 0

```

Query 17: Find the departments that sell all the items supplied by Hardware dept (and possibly more).

sales	Dept	Item	conditions
	P.G._d Hardware	[ALL._i,*] ALL._i	_d <> Hardware

Similar to Query 15, the set containment concept needs to be used. Here, we show only the SQL using the CONTAINS operator and omit the SQL without using it for brevity.

```

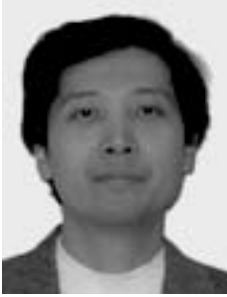
SELECT Dept
FROM sales
GROUP BY Dept
HAVING Dept <> 'Hardware' AND Item CONTAINS
      (SELECT Item FROM sales WHERE Dept = 'Hardware')

```


Author Biographies



Dongwon Lee received his B.S. from Korea University, Seoul, Korea in 1993, his M.S. from Columbia University, New York, USA in 1995, and his Ph.D. from UCLA, Los Angeles, USA in 2002, all in Computer Science. He is currently Assistant Professor in the School of Information Sciences and Technology at Penn State University, USA. His research interests include Web and XML Databases, Semantic Web, Digital Library, and Intelligent Information Systems. His homepage is at <http://nike.psu.edu/dongwon/>.



Wenlei Mao received his B.S. in Physics from Peking University, China in 1992, and his M.S. in Computer Science from the College of William and Mary, Virginia in 1996. He is currently working towards his Ph.D. degree in Computer Science at UCLA. His current research interests include Knowledge-based Information Retrieval and Intelligent Active Database Systems.



Henry Chiu received a B.S. degree in 2000 and M.S. degree in 2001 from the University of California at Los Angeles. He is currently a software engineer at IBM, developing database synchronization middleware for mobile devices.



Wesley W. Chu is a professor of Computer Science and was the past chairman (1988–1991) of the Computer Science Department at the University of California, Los Angeles. His current research interest is in the areas of distributed processing, knowledge-based information systems, and intelligent web-based databases. He was the conference chair of the 16th International Conference on Conceptual Modeling (ER'97). He is also currently a member of the Editorial Board of the Journal on Very Large Data Bases and an Associate Editor for the Journal of Data and Knowledge Engineering. Dr. Chu is a Fellow of IEEE.

Correspondence and offprint requests to: Dongwon Lee, School of Information Sciences and Technology, Penn State University, PA 16802, USA. Email: dongwon@psu.edu