TBE: Trigger-By-Example

Dongwon Lee **Wenlei Mao** **Wesley W. Chu**

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA

Email: {dongwon,wenlei,wwc}@cs.ucla.edu

Last Revised: August 3, 1999

Abstract

Triggers have been adopted as an important database feature and implemented by most major database vendors. Despite their diverse potential usages, one of the obstacles that hinder the triggers from its wide deployment is the lack of tools that aid users to create complex trigger rules in a simple manner. Although the majority of the users of triggers are DBAs or savvy end-users, writing trigger rules is still a daunting task.

On the other hand, QBE *(Query-By-Example) has been very popular as a user interface for creating queries in an interactive and intuitive manner since its introduction decades ago. It is being used in most modern database products in its disguised form. Since its underlying theory is based on the relational calculus, its expressive power is proved to be equivalent to that of SQL. Therefore, it is an ideal tool for novice users to create simple queries in visual fashion. At the same time, expert users do not have to compromise anything because* QBE *supports full capability to create complex queries.*

In this paper, we shall present a novel user interface for creating trigger rules, called TBE *(Trigger-By-Example), by marrying the triggers and* QBE *in a seamless fashion. The visual nature of the* TBE *makes writing trigger rules much easier. We show how trigger rules in the emerging SQL standard (SQL3) can be represented using the* TBE *with minimal introduction of new constructs. Further, an algorithm to translate from the* TBE *to SQL3 triggers is developed and illustrated along with examples. Finally, a preliminary implementation of the* TBE *is presented as a proof of the concept.*

UCLA-CS-TR-990029

Contents

1 Introduction

Triggers provide a facility to autonomously react to events occurring on the data, by evaluating a data-dependent condition, and by executing a reaction whenever the condition evaluation yields a truth value. Such triggers have been adopted as an important database feature and implemented by most major database vendors. Despite their diverse potential usages, however, one of the obstacles that hinder the triggers from its wide deployment is the lack of tools that aid users to create complex trigger rules in a simple manner. In many environments, the correctness of the written trigger rules is very crucial since the semantics encoded in the trigger rules are shared by many applications (known as *knowledge independence*) [16]. 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.

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 the domain relational calculus, its expressive power is proved to be equivalent to that of SQL that is based on the tuple relational calculus [2]. As opposed to SQL by which the user has to conform to the phrase structure strictly, the QBE user 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 [15].

In this paper, we propose to use the established QBE as a user interface for writing trigger rules. Since most trigger rules are complex combinations of SQL statements, by using the QBE as a user interface for triggers, the user may create only admissible trigger rules. The main idea is to use the QBE in a *declarative* fashion for writing the *procedural* trigger rules [6]. The main contributions of this paper are:

- Using the QBE as an interface for writing triggers.
- Developing the TBE scheme by extending the QBE with minimal introduction of new constructs for ECA model-based SQL3 triggers.
- Developing a translation algorithm from the proposed TBE to SQL3 triggers.

To ease discussion, we shall briefly remind SQL3 triggers and QBE in the following subsections.

1.1 SQL3 Triggers

In SQL3, *triggers*, sometimes called *event-condition-action rules* or *ECA rules*, mainly consist of three parts to describe the event, condition, and action, respectively. Since SQL3 is still evolving at the time of writing this paper, albeit close to its finalization, we base our discussion on the latest ANSI X3H2 SQL3 working draft [11]. The following is a definition of SQL3:

Example 1: SQL3 triggers definition.

<**SQL3-trigger**> ::= CREATE TRIGGER <**rule-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)

The QBE is a query language as well as a visual user interface. In the 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 kind of 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 the QBE convention, variable names are lowercase alphabets prefixed with "_", system commands are uppercase alphabets suffixed with ".", and constants are denoted without quote unlike SQL3. Let us see the QBE example. We use the following schema throughout the paper by default.

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

emp(**<u>Eno</u>, Ename, DeptNo, Sal)**
dept(**<u>Dno</u>, Dname, MgrNo)**

Then, 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
	_e	Tom		
		P.	_d	

dept	Dno	Dname	MgrNo
	_d		_e

The organization of this paper is as follows. Section 2 proposes our TBE for SQL3 triggers and discusses several related issues. A few complex SQL3 trigger examples are illustrated in Section 3. A translation algorithm from the TBE to triggers and its preliminary implementation are presented in Section 4 and Section 5. Section 6 illustrates potential applications of the TBE. Related works and concluding remarks are given in Section 7 and Section 8.

2 TBE: Trigger-By-Example

We propose to use the QBE as a user interface for creating trigger rules. Our tool is called the TBE, standing for Trigger-By-Example, implying that ours is in the same spirit as the classical QBE. The philosophy of the 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 in order to understand and use the whole language [15]. We attain the same benefits by using the QBE as an interface for creating trigger rules.

2.1 Difficulty of Expressing Procedural Triggers in Declarative QBE

Triggers in SQL3 are in nature procedural. As shown in Example 1, triggers action 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.

On the contrary, the QBE is a declarative query language. While writing a query, the user does not have to know if the first row in skeleton tables needs to be executed before the second row or not. That is, the order is immaterial. Also the 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, the 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 the QBE as much as possible.

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

2.2 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 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 (see Example 12 for instance). 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 for briefness.

2.3 Event-Condition-Action Triggers

SQL3 triggers use the ECA model. Therefore, triggers are represented by mainly three isolated E, C, A parts. In the `TBE`, each E, C, A part maps to the corresponding skeleton tables separately. To differentiate among three parts, three prefix flags, `E.`, `C.`, `A.`, are introduced. That is, in skeleton tables, table name is prefixed with one of these flags. The condition box in the QBE is also similarly extended. For instance, a condition statement ia specified in the `C.` prefixed skeleton table and condition box as depicted below.

C.emp	Eno	Ename	DeptNo	Sal

C.conditions

[1] SQL3 triggers definition in [11] leaves it implementation-defined whether the transaction, connection, or session statements shall not be generally contained in the action part or not.

2.4 Triggers Event Types

SQL3 triggers allow only the INSERT, DELETE, and UPDATE as legal event types. Coincidentally, the QBE has constructs `I.`, `D.`, and `U.` for each event type to describe the data manipulation query. The `TBE` uses these constructs to describe the trigger event types. Since the INSERT and DELETE always affect the whole tuple, not individual columns, `I.` and `D.` must be filled in the leftmost column of skeleton table. When the UPDATE trigger is described as to particular column, then `U.` is filled in the corresponding column. Otherwise, `U.` is filled in the leftmost column. Consider the following example.

Example 4: The first and second skeleton tables depict triggers that monitor the INSERT and DELETE events, respectively. The third one depicts the UPDATE event of the column `Dname` and `MgrNo`. Thus, changes occurring in other columns do not fire the trigger. The last one depicts the UPDATE event of any columns on the `dept` table. Note that all table names are prefixed with `E.` flags.

E.dept	Dno	Dname	MgrNo
I.			

E.dept	Dno	Dname	MgrNo
D.			

E.dept	Dno	Dname	MgrNo
		U.	U.

E.dept	Dno	Dname	MgrNo
U.			

Note that since SQL3 triggers definition limits that only a *single* event be monitored per *single* rule, there can not be more than one row having `I.`, `D.`, or `U.` flag unless multiple trigger rules are written together. Therefore, same trigger actions for different events (e.g., "abort when either INSERT or DELETE occurs") need to be expressed as separate trigger rules in SQL3 triggers.

2.5 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 of the input data. The *after* mode triggers execute after their event and are typically used to embed application logic [6]. In the `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 the `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.

2.6 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 the triggers, whether 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 the TBE.

1. *Using 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 value 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 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
AFT.U.R.				_s

E.conditions
NEW(_s) > OLD(_s) * 2

It is illegal to apply the `NEW()` or `NEW_TABLE()` to the variable defined on the DELETE event. Likewise 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()` the variable defined on the INSERT event. Likewise for the application of `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.

Example 6: 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	_d	_d	_o	_n

E.conditions
_n > _o * 2

We chose the approach using new built-in functions to introduce transition values into the `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 the 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 although one attribute emp.Ename is sufficient because no update occurs for this attribute).

2.7 The REFERENCING Construct

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

2.8 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 the `TBE` due to their procedural nature. Because their expressive power is beyond what the declarative QBE, and thus the `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. An example is:

A.statements
IF (X > 10) ROLLBACK;

2.9 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, the `TBE` uses an implicit agreement; like prolog, 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 that written at the bottom one.

2.10 Expressing Conditions in the 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 [1], SAMOS [9], Chimera [5]), however, provide filtering mechanisms that perform tests on event parameters (see [12], chapter 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 the `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 triggers languages who have both PF and GC types in future[3].

1. *Parameter Filter Type*: Since this type tests on 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 Section 2.6 are these types. In the `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 the `TBE`, this type is typically represented in the `C.` prefixed condition boxes. One such example is illustrated in Example 7.

Example 7: When an employee's salary is increased more than twice within the same year (a variable CURRENT_YEAR contains the current year value), record 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.		_c + 1	
	NEW(_n)	_c	CURRENT_YEAR

A.log	Eno	Sal
I.	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 condition as GC type, not PF type. Note that the headers of the sal-change and condition box have the `C.` prefixes.

3 Complex SQL3 Triggers Examples

In this section, we show a few complex SQL3 triggers and their `TBE` representations. These examples are adopted from [8, 16].

[3]By being able to differentiate PF and GC types, the user has more freedom to explicitly express trigger rules in a finer granularity. In fact, the `TBE` translation algorithm can determine if the given condition definition is of PF or GC type. Future implementation will include an optimization that automatically generates PF type conditions for the systems that support PF type.

3.1 Integrity Constraint Triggers

Triggers to maintain the foreign key constraint are shown below.

Example 8: 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 missing on purpose; 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 that how _e variable is used to join the emp and dept tables to find the department whose manager is just deleted. 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 9: When employees are inserted to the 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 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**
<u>FOR EACH ROW</u>
WHEN NOT EXISTS (SELECT * FROM **dept D** WHERE **D.Dno** = NEW.**DeptNo**)
  ROLLBACK

3.2 View Maintenance Triggers

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

Example 10: 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.Following is only for UPDATE event case.

Example 11: 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 the `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 actual synchronization.

Example 12: 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 that 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 to distinguish different trigger rules in rule generation time.

4 Generating Trigger Rules from the TBE

In this section, we discuss about the algorithm to generate trigger rules from the TBE. Ours is an extension of the algorithm by McLeod [10], which describes a translation from the QBE to SQL. Its input is a list of skeleton tables and the condition boxes and its output is a SQL query string. Let us denote the McLeod's algorithm as qbe2sql(<input>) and ours as tbe2triggers[4].

4.1 Algorithm Outline: tbe2triggers

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, condition or statement boxes are collectively called as *entry*.

1. *Preprocessing*: This step does two tasks: 1) reducing the TBE query to an equivalent, but simpler form (i.e., move the condition box entries that can be moved to the skeleton tables), and 2) partitioning the TBE query into distinct groups when multiple trigger rules are written in the query

[4] qbe2sql algorithm considers only SELECT statement, excluding INSERT, DELETE, UPDATE statements. Due to the nature of the triggers, TBE heavily uses such data modification statements. In this algorithm, we assume that McLeod's qbe2sql(<input>) algorithm can handle not only SELECT but also INSERT, DELETE, UPDATE statements. The extension is trivial. For details, refer ro [10].

together (i.e., Example 12). This can be done easily by comparing variables filled in the skeleton tables and collect those entries with the same variables being used into the same group. Then, the following steps 2, 3, and 4 are applied to each distinct group repeatedly to generate separate trigger rules.

2. *Build event clause*: Input all the `E.` prefixed entries. The "`CREATE TRIGGER` <rule-name>" clause is generated by the trigger name <rule-name> filled in the name box. By checking the constructs (e.g., `AFT.`, `R.`), system can determine the activation time and granularity of the triggers. Event type can also be detected by constructs (e.g., `I.`, `D.`, `U.`). If `U.` event is filled in individual columns, then "`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.`), then it is the "parameter filter (PF)" type conditions. Since SQL3 use the `WHEN` clause to represent this type, 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 the entry already contains `P.` command. This will result in creating "`SELECT` $table_1$`.*, ...,` $table_n$`.* FROM` $table_1$`, ...,` $table_n$" clause.

   (c) Gather all entries into <input> list and call qbe2sql(<input>) algorithm. Let the returned SQL string as <condition-statement>. For row-level triggers, create "`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 SQL3 format like step 3.(a).

   (b) Partition the entries into distinct groups. That is, gather entries with identical variables being used into the same group. Each group will have one data modification statement such as INSERT, DELETE, or UPDATE. Preserve the order semantics among partitioned groups such that 1) an entry appearing prior to another entry has higher order, 2) a group appearing prior to another group has higher order, and 3) a group having higher ordered entry than another group's entry has higher order.

(c) For each group G_i, call qbe2sql($< G_i >$) algorithm according to the order decided in the previous step. Let the resulting SQL string for G_i as <action-statement>$_i$. Note that statement box entries are not passed into qbe2sql algorithm since they are procedural. Instead, its contents 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 Implementation

A preliminary version of the `TBE` prototype is being implemented using jdk 1.2.1. Although the underlying concept is the same as what we have presented so far, we added several bells and whistles (e.g., context sensitive pop-up menu) for better human-computer interaction.

The main screen consists of two sections – one for input and another for output. The input section is where the user creates trigger rules by the QBE mechanism and the output section is where the interface generates trigger rules in the target trigger syntax. Further, the input section consists of three panes for event, condition, action, respectively. The main screen of the prototype is shown in Figure 1, where the query in Example 7 is shown.

A query wizard like feature (to show users all the available yet valid options at the particular time and have them select one) is introduced in the implementation via context-sensitive pop-up menus. For instance, the implementation first shows all available table names to the user when empty skeleton table is inserted. After the user picks one table, its attribute names appear in the skeleton table automatically. Also, when the right mouse button is clicked at some columns of the skeleton table, valid behavior options (e.g., insert example variable, insert transition variable) are listed in the pop-up menu to aid the user's selection. One such pop-up menu is shown in Figure 1 as well. After filling data in the input section, when the user clicks the down arrow button, the equivalent trigger rule is generated at the output section.

6 Applications

Not only is the `TBE` useful for writing trigger rules, but it can also be used for other applications with a few modifications. Two such applications are illustrated in this section.

6.1 Declarative Constraints in SQL3

SQL3 has the `ASSERTION` to enforce any condition expression that can follow `WHERE` clause to embed some application logic. The form of an assertion is:

CREATE ASSERTION **<assertion-name>** CHECK **<condition-statement>**

Note the similarity between the assertion and triggers syntax in SQL3. Therefore, a straightforward extension of the `TBE` can be used as a tool to enforce assertion constraints declaratively. In fact, since the

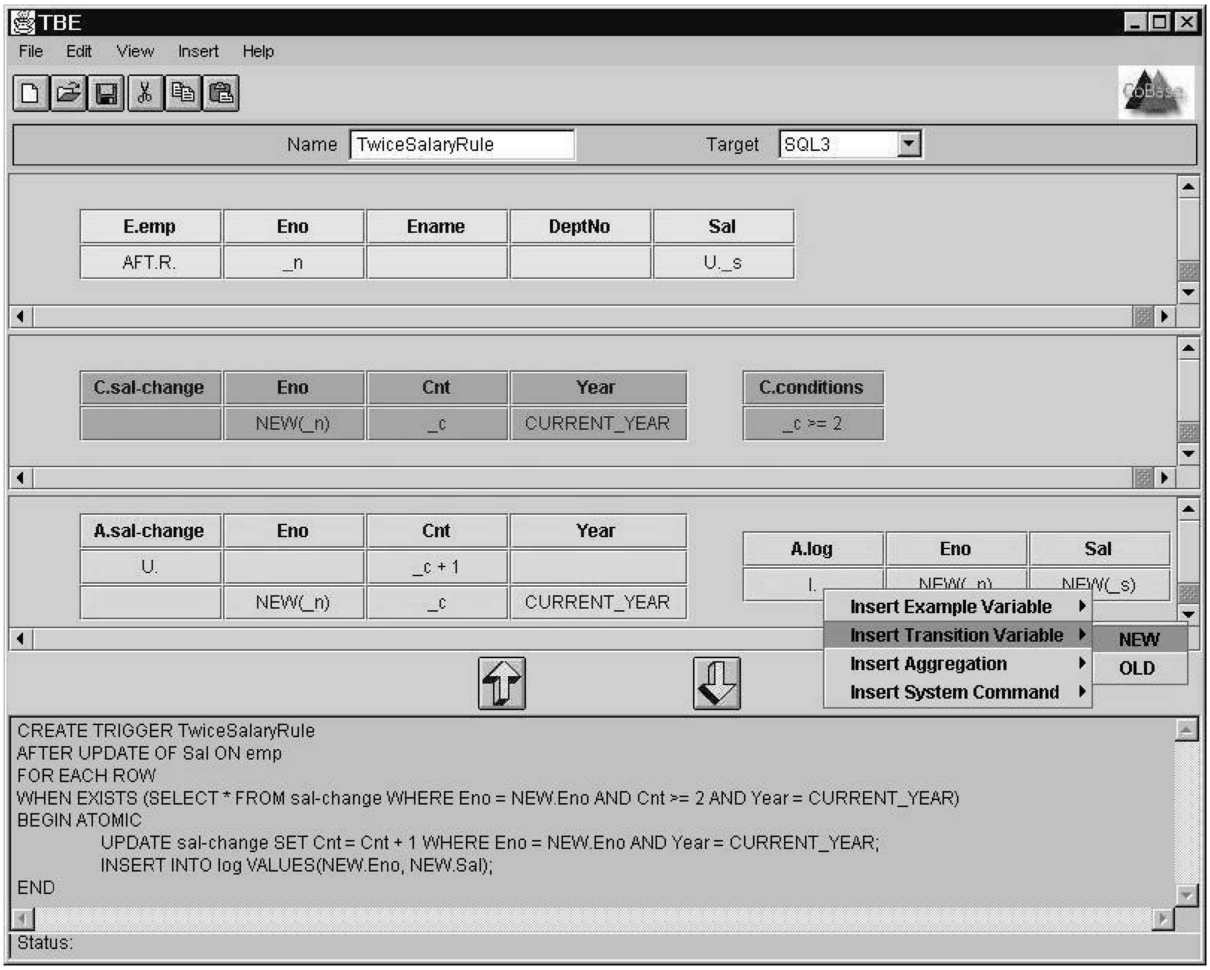

Figure 1: Main screen dump.

`ASSERTION` in SQL3 only permits declarative constraints, the `TBE` suits the purpose perfectly.

6.2 Universal Triggers Construction Tool

Although SQL3 is close to its final form, many database vendors are already shipping their products with their own proprietary triggers syntaxes and implementations. When multiple databases are used together or one database needs to be migrated to another, these diversities can introduce significant problems. To remedy this problem, one can use the `TBE` as a universal triggers construction tool. That is, the user creates triggers using the `TBE` interface and saves it as the `TBE`'s internal format[5]. When one database is changed to another, the user can simply reset one of the preference information of the `TBE` (e.g., from Oracle to DB2) to re-generate new trigger rules. Extending the `TBE` to support all unique features of diverse database products is not a trivial task. Nevertheless, we believe that retaining the visual nature of the triggers construction with the `TBE` can be quite useful in coping with heterogeneous database systems.

[5]This can be a modified domain relational calculus format or linearized `QBE` format [10].

7 Related Works

Past active database research has focused on active database rule language (e.g., [1]), rule execution semantics (e.g., [6]), or rule management and system architecture issues (e.g., [13]). In addition, research on visual querying has been done in traditional database research (e.g., [7, 15]). To a greater or lesser extent, all these research focused on devising novel visual querying schemes to replace data retrieval aspects of the SQL language. Although some has considered data definition aspects [3] or manipulation aspects, none has extensively considered the *trigger* aspects of the SQL, especially from the user interface point of view.

Other works (e.g., IFO_2 [14], IDEA [5]) 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 [5] 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. The TBE, on the other hand, tries to help users to *directly* design active rules with minimal learning.

Other than the QBE skeleton tables, *forms* have been popular building blocks for visual querying mechanism as well. For instance, [7] proposes the NFQL as a communication language between human and database system. It uses forms in a strictly nonprocedural manner to represent query. Other works using forms are mostly for querying aspect of the visual interface [3].

To the best of our knowledge, the only work that is directly comparable to ours is RBE [4]. Although RBE also uses the idea of the QBE as an interface for creating trigger rules, there are following significant differences:

- Since the TBE is carefully designed with SQL3 triggers in mind, it is capable of creating all the complex SQL3 trigger rules. Since RBE's capability is, however, limited to OPS5-style production rules, it cannot express, for instance, the subtle difference of the trigger activation time nor granularity.

- Since RBE focuses on building an active database system in which RBE is only a small part, no evident suggestion of the QBE as a user interface to trigger construction is given. On the contrary, the 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 the TBE implementation is a thin layer utilizing a translation from a visual representation to the underlying triggers, it is a loosely coupled with the database.

8 Conclusion

A novel user interface called TBE for creating triggers is proposed. TBE borrows the visual querying mechanism from the QBE and applies it to triggers construction application in a seamless fashion. An array of new constructs are introduced to extend the QBE to support triggers semantics and syntaxes properly.

In addition, a translation algorithm from the visual representation of the TBE to the underlying SQL3 triggers is developed. Furthermore, to prove the concept, a prototype is implemented and demonstrated the feasibility and benefits of applying the QBE to the trigger rule writing.

References

[1] R. Agrawal, N. Gehani, "Ode (Object Database and Environment): The Language and the Data Model", *Proc. SIGMOD*, Portland, Oregon, 1989.

[2] E. F. Codd, "Relational Completeness of Data Base Languages", *Data Base Systems, Courant Computer Symposia Series*, Prentice-Hall, 6:65-98, 1972.

[3] C. Collet, E. Brunel, "Definition and Manipulation of Forms with FO2", *Proc. IFIP Visual Database Systems*, 1992.

[4] Y.-I. Chang, F.-L. Chen, "RBE: A Rule-by-example Action Database System", *Software – Practice and Experience*, 27(4):365-394, 1997.

[5] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca, "Active Rule Management in Chimera", In J. Widom and S. Ceri (ed.), *Active Database Systems: Triggers and Rules for Active Database Processing*, Morgan Kaufmann, 1996.

[6] R. Cochrane, H. Pirahesh, N. Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Systems", *Proc. VLDB*, 1996.

[7] D. W. Embley, "NFQL: The Natural Forms Query Language", *ACM TODS*, 14(2):168-211, 1989.

[8] S. M. Embury, P. M. D. Gray, "Database Internal Applications", In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag, 1998.

[9] S. Gatziu, K. R. Dittrich, "SAMOS", In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag, 1998.

[10] D. McLeod, "The Translation and Compatibility of SEQUEL and Query by Example", *Proc. Int'l Conf. Software Engineering*, San Francisco, CA, 1976.

[11] J. Melton (ed.), "(ANSI/ISO Working Draft) Foundation (SQL/Foundation)", *ANSI X3H2-99-079/WG3:YGJ-011*, March, 1999. (ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-foundation-wd-1999-03.pdf)

[12] N. W. Paton (ed.), "Active Rules in Database Systems", *Springer-Verlag*, 1998.

[13] E. Simon, A. Kotz-Dittrich, "Promises and Realities of Active Database Systems", *Proc. VLDB* 1995.

[14] M. Teisseire, P. Poncelet, R. Cichetti, "Towards Event-Driven Modelling for Database Design", *Proc. VLDB*, 1994.

[15] M. M. Zloof, "Query-by-Example: a data base language", *IBM System J.*, 16(4):342-343, 1977.

[16] C. Zaniolo, S. Ceri, C. Faloutsos, R. R. Snodgrass, V.S. Subrahmanian, R. Zicari, "Advanced Database Systems", *Morgan Kaufmann*, 1997.

A Appendix

McLeod's original paper 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. First, examples are based on following schema:

emp(**Ename, Sal, Mgr, Dept**)
sales(**Dept, Item**)
supply(**Item, Supplier**)
type(**Item, Color, 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 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	Color
	P.	red

SELECT **Item**
FROM **type**
WHERE **Color** = **'red'**

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

sales	Dept	Item
	P.	_i

supply	Item	Supplier
	_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.	_e1	_m
	_m	_e2	

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	pen
	_d	pencil

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

In QBE, same query can be expressed using 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	pen
	P._d2	pencil

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

In QBE, same query can be expressed using 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, supplier pairs such that the department sells an item that the supplier supplies.

sales	Dept	Item
	P._d	_i

supply	Item	Supplier
	_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	Color
	P.	¬ green

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

All following QBE and SQL expressions are equivalent.

type	Item	Color
	P._i	
¬	_i	green

type	Item	Color
	P._i	
	¬ _i	green

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

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

Query 9: Find the departments each of which sells items supplied by parker and bic.

sales	Dept	Item
	P._d	_i1
	_d	_i2

supply	Item	Supplier
	_i1	parker
	_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
	P._d	[_i1,_i2]

supply	Item	Supplier
	[_i1,_i2]	[parker,bic]

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

sales	Dept	Item
	P._d	_i

supply	Item	Supplier
	_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 translation 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
	_s	P.G._d

sales	Dept	Item
	_d	pen

conditions
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
	P.	P._d

sales	Dept	Item
	G._d	_i

conditions
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 same as the total number of items of all the suppliers.

sales	Dept	Item
	P.G._d	_i1

supply	Item	Supplier
	_i1	
	_i2	

conditions
CNT.UNQ.ALL._i1 = CNT.UNQ.ALL._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
	P.G.	[ALL._i,*]

supply	Item	Supplier
	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
	P.G.	ALL._i

supply	Item	Supplier
	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
	P.G._d	[ALL._i,*]
	Hardware	ALL._i

conditions
_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 briefness.

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