Timer-Driven Database Triggers and Alerters

Eric N. Hanson and Lloyd X. Noronha
301 CSE
CISE Department
University of Florida
Gainesville, FL 32611
352-392-2691
hanson@cise.ufl.edu

TR-99-011
August 5, 1999

Abstract[†]

This paper proposes a simple model for a timer-driven triggering and alerting system. Such a system can be used with relational and object-relational databases systems. Timer-driven trigger systems have a number of advantages over traditional trigger systems that test trigger conditions and run trigger actions in response to update events. They are relatively easy to implement since they can be built using a middleware program that simply runs SQL statements against a DBMS. Also, they can check certain types of conditions, such as "a value did not change" or "a value did not change by more than 10% in six months." Such conditions may be of interest for a particular application, but cannot be checked correctly by an event-driven trigger system. Also, users may be perfectly happy being notified once a day, once a week, or even less often of certain conditions, depending on their application. Timer triggers are appropriate for these users. The semantics of timer triggers are defined here using a simple procedure. Timer triggers are meant to complement event-driven triggers, not replace them. *We challenge the database research community to developed alternate algorithms and optimizations for processing timer triggers, provided that the semantics are the same as when using the simple procedure presented here.*

1 Introduction

The trigger systems in current database products are synchronous. In other words, trigger conditions are checked inside of update transactions, and trigger actions run inside the triggering transaction. This has the advantage that triggers can be used to enforce integrity constraints, and that triggered actions that update the database will only be carried out if the triggering transaction commits. However, checking trigger conditions inside update transactions effectively limits the total number of triggers that can be specified for performance reasons. Moreover, when triggers are used as alerters (rules to notify humans when changes of interest have occurred in the data), checking trigger conditions inside update transactions is overkill in some cases. It may be perfectly reasonable to be notified only once an hour, once a day, or once a week when an interesting change occurs. This is especially true when the change of interest is based on an aggregation, such as a sum, average, or count.

Also, with a timer trigger, it is possible to check conditions such as "did not change" or "did not change by more than 10% in six months." These conditions can be true even if the relevant data is not updated, so they cannot be checked correctly in a trigger system driven by update events. Some type of timer-driven polling mechanism is required.

Scalable trigger processing has recently been identified as an important research topic by a group of leading database researchers [1]. Checking trigger conditions infrequently, when this is acceptable for the application, is an excellent way to improve the scalability of a trigger system. Many more timer-driven

[†] This research was supported in part by grants from the Defense Advanced Research Projects Agency and Informix Software, Inc.

triggers than synchronous triggers can be handled in a high-update-rate database, assuming that timer expiration for each trigger is not too frequent. Obviously, checking thousands of trigger conditions every second using a timer-driven mechanism is a recipe for disaster. But a trigger system can check thousands, perhaps millions of trigger or alerter conditions daily and perform well.

Timer driven triggers similar in some respects to the ones discussed here are not entirely new. Batch processing systems that can automatically create reports at scheduled times and email them to users or print them for delivery have been around for years. In addition, a system that lets users create timer-based triggers on Web pages is commercially available from NetMind [9]. The system that is probably most closely related to the discussion in this paper is Categoric Alerts, a commercial timer-driven alerting system for relational databases [2]. Another commercial system from Comshare, Comshare Detect and Alert, also has some timer-driven trigger capability [4]. Detect and Alert is oriented towards long-term change detection, e.g. to detect if profits have increased by 10% or more from the same quarter in the year before for a business.

The discussion presented here is not meant to advocate that timer-driven triggers should completely replace conventional synchronous triggers, such as SQL-3 triggers [3], or asynchronous triggers that are checked immediately after update transactions commit, as in the TriggerMan system [7]. Rather, timer-driven triggers complement them. Developers can choose timer driven triggers when they are useful for their application.

This paper discusses a timer-driven triggering mechanism, which is useful in situations where only periodic trigger condition checking is required, and in situations where it is simply too expensive to check trigger conditions within update transactions. One of the advantages of the approach is its simplicity. It can be implemented in middleware over a DBMS, making use of the DBMS query processor for trigger condition testing. The middleware can run in an external process, inside an object-relational DBMS extension module (e.g. an Informix DataBlade), or in a combination of the two. The remainder of this paper discuss the proposed trigger language, trigger processing architecture, and system implementation.

2 Trigger Language

We propose a timer-driven trigger mechanism based on the following trigger language. First, database tables are defined as data sources to the trigger processor. The trigger processor knows the schema of each data source table, and knows which column or columns make up the primary key of each data source. In the command syntax notation given in this paper, items in curly brackets are optional. Items in angle brackets will be more fully specified later.

2.1 Timer Trigger Creation

A timer-driven trigger can be created with the syntax shown below.

```
create timer trigger <triggername> {in <triggerset>}
from <from-clause>
{on event}
{when <when-qual>}
[ check every <time-specification> | check using <calendar-specification> ]
{initialize {immediately | on first timer expiration}}
{for each {row | set}}
do <action>
```

The **from** clause of the **create timer trigger** command can contain either a parenthesized SQL **select** statement, the name of a data source, or the name of a view defined on one or more other data sources or views.

Views can be created on data sources using the normal SQL **create view** command, augmented with a **primary key** clause. This command has the following syntax:

```
create view <viewname> as
  select <target-list>
```

from <from-list>
{ where <view-qual> }
{ group by <group-attr-list> }
{ having <view-having-qual> }
{ primary key (<attr-list>) }

If the **from** clause of the view definition contains only one table or view name, and the **select** in the view definition contains the key of this table or view in the target list, then a primary key clause does not have to be specified. Instead, the key is inferred to have the same attributes as the source table or view. If the view is an aggregate query with a **group by** clause, then its key is the set of attributes specified in the **group by** clause. If it is not possible to infer the key, then an error message is signaled and the **create trigger** command is refused. If it turns out that the attributes for a trigger's view do not form a key, a run-time error may be detected during trigger condition processing, the trigger will be deactivated, and an error message will be logged.

The **on** clause allows the user to check to see whether a row was inserted into, deleted from, or updated in the view or data source given in the **from** clause since the last time the trigger's timer expired. We'll discuss timer expiration more below. The event may be **insert**, **delete, update** or **update(<attribute-list>)**. If there is no **on** clause, then every time the trigger's timer expires, all rows in the trigger's data source or view are retrieved, and the trigger's action is executed for those rows.

If and only if the trigger has an **on update** clause, a Boolean expression can appear in the **when** clause. This expression may refer to any field retrieved by the view of the trigger. The notation **old** may be used in the **when** clause before any field name. For example, old.salary would be the old value of the salary field retrieved by the view. If the **when** clause does not appear for an **on update** trigger, it defaults to TRUE.

The **check every** and **check using** clauses specify when timers expire for the trigger. If the trigger has a **check every** <time-units> clause, then the timer expires after <time-units> of time have passed. The trigger's timer is then reset to go off <time-units> later.

If the trigger has a **check using** <calendar-specification> clause, then the trigger's timer expires at the time points specified by the calendar. The details of calendar specification are beyond the scope of this paper. However, a calendar can be specified by using its name, or by specifying a literal calendar expression in the place of <calendar-specification>. A separate **create calendar** command is used to create a calendar and give it a name. A <calendar-specification> can be used to create a trigger whose condition is checked every Monday, Wednesday and Friday at 5:00PM, for example.

The **initialize** clause is optional. The default value for this clause when it does not appear is **immediately**. The meaning of this clause will be defined later.

The **for each** clause is also optional. Its default value is **for each row**. If the timer trigger is a **for each row** trigger, then the action of the trigger is run once for each qualifying row. If the timer trigger is a **for each set** trigger, then the action is run once for the entire set of qualifying rows. The system is not required to process the rows in any particular order in a **for each row** trigger. In a **for each set** trigger, the system is not required to order the rows within the set in any particular way.

The **do** clause contains the trigger action. The action can contain a command in the trigger system's command language [7], or a **begin ... end** block containing a sequence of commands.

The semantics of the language are defined by a procedure given below. This procedure can be used to implement the language, but that is not required. The only requirement is that the results of timer-driven trigger processing must be *as if this procedure was used*. This leaves open opportunities for optimization.

2.2 Trigger Processing Procedure

The **view** V of a timer-driven trigger is defined as the data source, view, or **select** statement given in its **from** clause. The behavior for all cases, including no **on** clause, **on update, on delete,** and **on insert** is given below. In the following algorithms, it is implicitly assumed that the initialization and timer expiration procedures set the timer of the trigger to go off at the next appropriate time.

2.2.1 Algorithm NoOnClause

We restate the no **on** clause case as an algorithm for completeness. The behavior of timer triggers with no **on** clause is defined by the following procedure:

Initialization

Do nothing.

Timer Expiration

Run a query to retrieve the contents of the trigger's view. Run the trigger action for the data retrieved. More specifically, if the trigger is a **for each row** trigger, run the trigger's action once for each row retrieved. If it is a **for each statement** trigger, run the trigger's action once if any data is retrieved, and don't run the trigger's action if no data is retrieved. This applies to other types of timer triggers as well when we say "run the trigger action."

2.2.2 Algorithm OnUpdate

The behavior of timer driven triggers which have an **on update** event clause is defined according to the following procedure.

Initialization

If the trigger is an **initialize immediately** trigger, retrieve the current contents of the view and store them in TEMP1, and set the timer to go off at the appropriate time. If it is an **initialize on first timer expiration** trigger, then set the timer to go off at an appropriate time, and when the timer goes off the first time, retrieve the current contents of the view into TEMP1.[1]

Timer Expiration

After initialization, when the timer for the trigger goes off, perform the following steps:

1. Retrieve the current contents of V and store the result in TEMP2.
2. Let the **when** clause condition of the timer-driven trigger be called W. Run the following query, and run the trigger action for the values retrieved.

```
select *
from TEMP1, TEMP2
where TEMP1.key=TEMP2.key
and W
and (TEMP1.attr1 <> TEMP2.attr1 or          -- At least one attribute was updated.
     TEMP1.attr2 <> TEMP2.attr2 ... or
     TEMP1.attrN <> TEMP2.attrN)
```

3. Delete TEMP1.
4. Rename TEMP2 to TEMP1.[2]

If the trigger's **on update** clause specifies an attribute list of the form (attr_i1, attr_i2, ... attr_iK), rather than no attribute list, step 2 above is modified. In that case, the final **and** term is replaced by:

---

[1] For example, if the trigger is a "check every day at 2am" trigger with an **initialize on first timer expiration** clause, and the trigger is created at 3pm, then its view's contents would first be retrieved at 2am the next evening, and stored in TEMP1.

[2] Not all SQL databases support a table rename operation. However, the timer driven trigger system can simulate a rename operation by saving a new table name in its catalogs and logically associating it with TEMP1.

(TEMP1.attr_i1 <> TEMP2.attr_i1 or
TEMP1.attr_i2 <> TEMP2.attr_i2 ... or
TEMP1.attr_iK <> TEMP2.attr_iK)

The procedure as given defines the semantics of timer driven trigger processing for timer triggers with an **on update** clause. Again, the timer trigger system can use this algorithm, or another algorithm, as long as the system behaves *as if this algorithm was used.*

The way Algorithm OnUpdate is defined, a trigger cannot fire for a tuple unless that tuple exists when the timer expires and at the previous time the timer expired. When we say "tuple" we mean "tuple with the same primary key." In other words, for an **on update** trigger to fire for a tuple, the tuple must have been updated, at least logically. It may have been deleted and reinserted, but that is a logical update from the point of view of the timer trigger system.

We believe that it is important for **on update** timer triggers to behave this way to make it easy for users to understand how the system works. The **on insert** and **on delete** timer triggers also behave in a way that should be relatively easy for users to understand.

2.2.3 Algorithm OnInsert

When a timer trigger has an **on insert** clause, the procedure that defines its behavior is slightly different. It is defined as follows.

Initialization

The initialization procedure is the same as for **on update** triggers.

Timer Expiration

When the timer goes off, these steps are executed:

1. Retrieve the current contents of V and store the result in TEMP2.
2. Form TEMP3 as follows (the - sign represents the set difference operation):

   TEMP3 = TEMP2 - TEMP1
3. Run the trigger action for the values in TEMP3.
4. Delete TEMP1.
5. Rename TEMP2 to TEMP1.
6. Delete TEMP3.

For an **on insert** timer trigger to fire for a tuple, that tuple must not have existed in the view the previous time the timer expired, and must exist at the current timer expiration.

2.2.4 Algorithm OnDelete

When a timer trigger has an **on delete** clause, the procedure that defines its behavior is defined this way:

Initialization

The initialization procedure is the same as for **on update** triggers.

Timer Expiration

When the timer goes off, these steps are executed:

1. Retrieve the current contents of V and store the result in TEMP2.
2. Form TEMP3 as follows (the - sign represents the set difference operation):

   TEMP3 = TEMP1 - TEMP2
3. Run the trigger action for the values in TEMP3.

4. Delete TEMP1.
5. Rename TEMP2 to TEMP1.
6. Delete TEMP3.

The only difference from the previous procedure is that step 2 changed.

For an **on delete** timer trigger to fire for a tuple, that tuple must have existed in the view when the timer expired previously, and must not exist now.

3 Examples

Examples of triggers created with the language just described are given here. Consider the following schema for a retail store checkout database:

```
checkout(cno,date,time,sno)
lineitem(lino,cno,pno,qty,unitprice)
product(pno,description,category,unitprice,qoh)
store(sno, address, phone, manager_name, manager_email)
```

For each checkout, there are multiple line items. The unitprice attribute represents the price of a product. Unitprice is copied from a product table row into a lineitem row when someone purchases one or more of that product. The qoh field of product stands for quantity on hand.

As a simple first example, suppose user Bob wants to be notified by email whenever the quantity on hand of the item '25 oz. claw hammer' is less than 10. This trigger does not specify any update event, so we can specify it with no **on** clause, like this:

```
create timer trigger notify_bob
from (select pno, qoh
        from product
        where description = '25 oz. claw hammer'
        and qoh < 10)
check every day
do email('Bob@acme.com', '25 oz. claw hammer qoh = :qoh')
```

The use of the notation :qoh in the **do** clause of the trigger indicates that :qoh is supposed to be replaced using macro expansion.

A similar but slightly more sophisticated example is shown next. This example is a transition trigger, since it refers both to the old and current state of data. This trigger notifies Bob whenever the quantity on hand of item '25 oz. claw hammer' was 10 or more, but then drops below 10, checking this condition daily.

```
create timer trigger notify_bob2
on update(qoh)
from (select pno, qoh
        from product
        where description = '25 oz. claw hammer')
when qoh < 10 and :old.qoh >= 10
check every day
do email('Bob@acme.com', '25 oz. claw hammer qoh = :qoh; old qoh = :old.qoh')
```

A trigger that is based on a more sophisticated query, involving an aggregate, is shown below. This trigger notifies the manager of a store whenever there is more than a 30% jump in sales for the week, compared with the previous week:

```
create timer trigger thirty_pct_sales_jump
on update
from (select store.sno as sno, manager_email, sum(qty*unitprice) as sales
        from product, lineitem, checkout, store
```

```
        where product.pno=lineitem.pno
        and lineitem.cno=checkout.cno
        and store.sno=checkout.sno
        and checkout.date >= CURRENT_DATE - 7
        group by sno, manager_email)
    when sales > 1.30*old.sales
    check every week beginning 'Sunday at 2:00 AM'
    do email('manager_email',
            'total sales jump\:  sno = :sno, this week = :sales, last week = :old.sales')
```

Notice that no primary key clause is needed for the select statement in the above trigger because the **group by** attributes are automatically inferred to be the primary key. This trigger also illustrates the use of the CURRENT_DATE expression, which evaluates to the current date, as allowed in SQL [5]. Similarly, CURRENT_TIME and CURRENT_TIMESTAMP can also be used. The backslash used in the **do** clause of the trigger is the escape character and is used to allow the : to appear in the output without having macro processing applied to it.

Another example of an alerter is this one, based on a the table emp(eno,name,sal,birthdate,email_addr). It alerts Bob if his salary is different from that of Fred:

```
create timer trigger bob_fred
from (select e1.sal as e1_sal, e2.sal as e2_sal
        from emp e1, emp e2
        where e1.name = 'Bob' and 'e2.name = 'Fred')
when e2_sal != e1_sal
check every day
do email('bob@acme.com', 'your salary is different from that of Fred')
```

A useful type of trigger checks if new data has entered a view. For example, suppose that Fred wants to know whenever new rows are entered in the product table with category='hardware'. This could be done with the following view and trigger:

```
create view hardware as
select *
from product
where category='hardware'

create timer trigger notify_fred_new_hw
from hardware
on insert
check every day
do email('fred@acme.com', 'new hardware product :pno, :description')
```

Suppose an executive Fritz wants to know whenever a store is deleted, but is satisfied if he is notified of this within a week. That could be done in the following way:

```
create timer trigger store_delete_watch
from store
on delete
check every week
do email('fritz@acme.com', 'store :sno at :address was deleted')
```

When data sources or views have timestamp or date attributes, by using the CURRENT_TIMESTAMP, CURRENT_TIME, or CURRENT_DATE functions, it is possible for the user to create triggers that can be processed more efficiently by the system. For example, suppose that a manager Pam wants to know about all purchases of the item described as 'red rover 3500 lawn mower' in the last 7 days, and wants to be notified weekly. This could be done with the following trigger:

```
create timer trigger mower_watch
from (select checkout.*, lineitem.*
        from checkout, lineitem, product
        where checkout.cno=lineitem.cno
        and lineitem.pno=product.pno
        and product.description = 'red rover 3500 lawn mower'
        and checkout.date >= CURRENT_DATE -7)
check every week
do email ('pam@acme.com',
          'red rover 3500 mower purchased on :checkout.date, cno :checkout.cno')
```

The nested select statement in the above trigger may not be expensive to run be cause the conditions

- product.description = 'red rover 3500 lawn mower', and
- checkout.date >= CURRENT_DATE -7

are reasonably selective. The value of trigger view definitions that refer to the CURRENT_DATE, CURRENT_TIME or CURRENT_TIMESTAMP is that often, no **on** clause must be used. The primary benefit of this is that it is not necessary to store an old copy of the view.

4 Timer Trigger System Architecture

The timer driven trigger mechanism can be implemented as a multithreaded server process that communicates with a DBMS. The communication mechanism can be ODBC [8] for portability, or a proprietary DBMS API for higher performance. Trigger system applications can communicate with the system via the trigger system API. These applications can provide a user interface to the timer trigger system, allowing timer triggers to be created, deleted, activated, deactivated, and so forth. They can also receive notifications when triggers fire. A diagram of the timer trigger system is as follows:

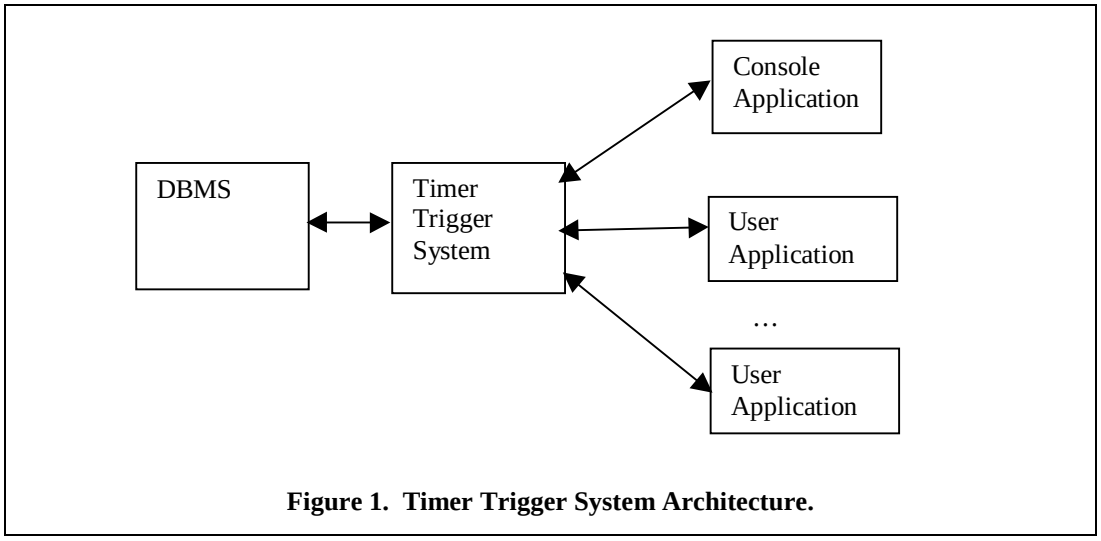

Figure 1. Timer Trigger System Architecture.

The Timer Trigger System (TTS) logic can be completely contained within the TTS process. Alternatively, it can be partitioned so that some of it appears inside the DBMS either in stored procedures or in user-defined routines (UDRs) in an object-relational DBMS extension module [12], and the rest remains in the TTS process. The simplest implementation is to keep all the logic in the TTS process. Partitioning the logic is more complex, but can give higher performance. In the next section, we describe the implementation of a non-partitioned implementation, where all the logic is in the TTS process.

5 Implementation Proposal

A timer trigger system can be implemented using the following basic structure. The trigger system catalogs can be stored as tables in a database maintained by the DBMS. The following relations are maintained:

1. **trigger:** to hold trigger definitions,
2. **view:** to hold view definitions, and
3. **data source:** to hold the names of tables that the trigger system can access.

When the TTS process boots, it reads these catalogs and builds a trigger wakeup time list to be managed by the wakeup thread (described below). Entries in the wakeup time list are small objects of the form <wakeup time, trigger ID>. Each entry does not take much memory. Hence, it is feasible to keep wakeup times for millions of triggers on this list.

A **timer trigger cache** and a **view cache** are maintained in the server address space. These hold main-memory objects describing recently accessed triggers and views. A trigger or view object can be built from its corresponding catalog row.

The TTS process is a multi-threaded process. It runs the following threads:

1. **Wakeup thread**: This thread maintains the wakeup timers for timer-driven triggers. When a timer goes off, it initiates processing for the triggers that require work to be done at that wakeup time.
2. **Worker threads**: A pool of worker threads run, maintaining open database connections to reduce connection overhead. When instructed by the wakeup thread, they check trigger conditions and run trigger actions. All checking of trigger conditions is done using SQL statements.
3. **Listener thread**: This thread listens for requests from applications. When it receives a request, it passes it to a worker thread to be processed.

6 Possible Optimizations

A number of alternate algorithms can be used that may speed up timer trigger processing. An optimizer could be developed to select among alternative algorithms, thus speeding up processing of timer triggers.

There are several cases to consider, depending on whether the trigger is an **on insert**, **on delete**, or **on update** trigger, whether it has no **on** clause, whether it has a transition condition in its **when** clause, and potentially other criteria. A trigger is said to have a transition condition in its **when** clause when it contains the symbol **old**. If the trigger's **when** clause does not refer to any prior state of the view, but only the latest state, then its **when** condition is not a transition condition. The following discussion presents alternative, potentially faster algorithms for the different cases just identified. It appears that there are many other possible algorithms. *We challenge the database research community to find them, and to develop optimization strategies for choosing among them!* A legal alternative algorithm must not change the semantics of timer triggers defined in this paper.

6.1 On update, partial transition condition

In the case of **on update** timer triggers, if the **when** clause does not contain a transition condition, or it is a conjunction of predicates, only some of which are transition conditions, then an optimization is possible. It is still necessary to maintain an old copy of the view to see if data has changed. However, the trigger can be modified to potentially reduce the size of the view that must be stored [10]. Consider the following trigger:

```
create timer trigger T
from (select * from <from-list> where <view-qual>)
on update
when P
do ...
...
```

Suppose P can be written as:

P = P' and P_{trans}

where P' does not contain any **old** fields and P_{trans} does. Then we can re-write T as:

```
create timer trigger T
from (select * from <from-list> where <view-qual> and P')
on update
when P_trans
do ...
...
```

This rule applies even if P does not refer to **old** fields at all. In that case, P' is P and P_{trans} is TRUE.

6.2 On update, transition condition

In the case of an **on update** condition with a real transition condition (one that contains :old), some additional optimizations are possible compared with the simple algorithm OnUpdate presented in section 2. Here, we illustrate some of these possible optimizations via an example.

6.2.1 Avoiding the Need to Test for Change

For certain **when** conditions, it is possible for the system to know that if the **when** condition is true for a row, the row definitely changed. This eliminates the need to test explicitly to see if the row changed. For example, consider this trigger:

```
create timer trigger salChange
from emp
on update
when sal > old.sal
check every day
do email (emp.email_addr, 'your salary changed from :old.sal to :new.sal')
```

In this case, the condition "sal > old.sal" cannot be true unless sal was updated. It is possible to write a simple theorem prover to identify cases like this. For example, if the following types of conditions are true, that implies the tuple changed:

```
attr <> old.attr
attr > old.attr
attr < old.attr
attr > CONSTANT * old.attr where CONSTANT >= 1
```

When conditions such as these are present, the test to make sure a qualifying tuple changed can be eliminated from the query in step 2 of Algorithm OnUpdate from section 2.2.2. The query in step 2 will become the following:

```
select *
from TEMP1, TEMP2
where TEMP1.key=TEMP2.key
and W
```

This will speed up execution of the query, significantly reducing the CPU time required. It is not necessary to identify all such cases because the original algorithm will work properly if this optimization is not used. Just identifying the common cases will suffice. After applying the optimization shown, other optimizations can then be used if applicable.

6.2.2 Using Instantaneous Triggers to Limit the Scope of Tests

We classify regular triggers and asynchronous triggers (like those in TriggerMan) as "instantaneous" triggers. If the database being monitored supports instantaneous triggers, they can be used in some cases to speed up processing of timer-driven triggers.

Consider the salChange trigger above. Since the condition in the **when** clause is sal > old.sal, we know that unless a particular row changed for an employee, there is no way for this trigger to fire for that employee. By using synchronous triggers, the set of emp row keys (eno values) for those rows updated, inserted and deleted in the last 24 hours can be identified and saved in a table S(eno). If the number of rows in this table at the end of each 24 hour period is normally much less than the number of rows in emp, then this may be a useful optimization.

In general, this type of optimization becomes more useful the more frequently the timer expires.

For this example, when the timer expires at the end of the day, the following steps are performed:

1. Find all emp tuples inserted or updated in the last day and put them in TEMP2:

   TEMP2 = $\pi_{emp.*}$ (emp JOIN S)

2. Find all old values of emp tuples deleted or updated in the last day and put them in TEMP3:

   TEMP3 = $\pi_{TEMP1.*}$ (TEMP1 JOIN S)

3. Run the trigger action for the values retrieved by the following expression. Only new/old pairs for updated tuples will be retrieved.

   $\sigma_{TEMP2.sal > TEMP3.sal}$(TEMP2 JOIN$_{eno}$ TEMP3)

4. Remove deleted and old updated tuples from TEMP1:

   TEMP1 = TEMP1 - TEMP3

5. Add inserted and new updated tuples to TEMP1 to bring it to the current state.

   TEMP1 = TEMP1 $\cup$ TEMP2

6. Empty TEMP2, TEMP3, and S.

If an index exists on the key column of emp (and TEMP1 if it is large) then even if emp is very large, the above steps will be relatively fast if S is small. A more detailed discussion of this type of optimization is given in [10]. Additional research is needed to help make an optimization decision regarding whether to use a brute force approach, or an alternate approach such as this one.

For many triggers, it is not necessary to save all the attributes of the view in the TEMP1 table. Only the primary key attribute(s) of the view, those attributes that explicitly appear in the **when** clause of the trigger using the view, and those view attributes that appear in the trigger action must be stored in TEMP1. In the above example, only eno, sal and email_addr must be stored in TEMP1.

Techniques similar to the one just shown, which can be used for **on insert** and **on delete** timer triggers, are described in [10].

6.3 Exploiting shared sub-expressions

The timer trigger model we have described offers numerous opportunities to increase performance by sharing the work of evaluating sub-expressions. At the simplest level, when multiple triggers are defined with identical conditions (including the **from, on** and **when** clauses if present), then the trigger condition can be checked just once for all of them. The qualifying tuples can be passed to each trigger action for execution. Moreover, at most one copy of the view for the trigger must be maintained.

More sophisticated techniques to exploit shared sub-expressions are also possible. For example, two **on update** triggers defined on the same view but with different **when** clause conditions could share the same stored view. It may well be possible to develop other techniques for sharing sub-expressions, such as identifying when one view contains another view.

Shared sub-expressions can also be exploited when running queries. When timers go off every hour, day, or week, month, or quarter, large batches of queries must be run. These queries are *known in advance*. Hence, the numerous existing techniques developed for multiple query optimization can be used to speed up processing of these groups of queries.

6.4 Using Existing View Maintenance Algorithms

A large number of view maintenance algorithms have been proposed [6]. Existing view maintenance algorithms can be used to improve the performance of timer trigger systems. Any time a view maintenance algorithm needs to have a copy of *both* the old and current states of a view, e.g. as in algorithm OnUpdate of section 2.2.2, we can use a view maintenance algorithm as follows.

1. At initialization time, run a query to retrieve the view into TEMP1.
2. Immediately copy TEMP1 to TEMP2.
3. DO UNTIL the timer trigger is deleted or deactivated
    a. Using the view maintenance algorithm, keep TEMP2 up to date at all times.
    b. The next time the timer expires, TEMP1 contains the old state of the view and TEMP2 contains the current state.
    c. Perform work specific to this type of timer trigger.
    d. Empty TEMP1 and copy the contents of TEMP2 into TEMP1.

Step c needs to read from TEMP2; hence it needs to lock TEMP2 in exclusive mode. If this will deny update access to the database to on-line users and that is not acceptable, then an alternative must be found. It should be possible to queue updates to TEMP2 temporarily and apply them after step d.

The advantage of this technique is that we can insert a (potentially complex) view maintenance algorithm into a timer trigger system as a subroutine. If the view maintenance algorithm is treated as an abstraction, the complexity of the timer trigger processing algorithm will not increase significantly.

The real challenge is deciding *when* to use this algorithm. It will not always be better to use view maintenance than to re-materialize the view when needed. Future research on optimization strategies to choose among alternative timer trigger processing algorithms may bear fruit.

7 Preventing Unwanted Repeated Execution

It is useful to prevent triggering for the same tuple key value repeatedly. Triggers with **on insert** conditions already allow triggers to fire the first time a tuple matches a condition, i.e. when the tuple first enters a view. A more sophisticated mechanism for preventing unwanted repeated execution may also be desirable. A useful approach may be to allow the user to specify that a trigger should fire for a particular tuple key value K times in a row. Here, "K times in a row" means at K successive times that the timer expires. Another useful method may be to prevent a rule from firing for a particular tuple key value unless a "reset period" had passed since the previous firing for that key. Future research could first focus on developing methods for preventing unwanted repeated execution that are useful to the users of a timer trigger system. Then, efficient strategies could be developed to implement these methods.

8 Security, Authorization, and Quota Enforcement

As future research, security, authorization and quota enforcement mechanisms need to be developed for timer trigger systems. Clearly, the user that created a timer trigger should not be able to use the trigger to see data in the underlying DBMS that he or she does not have rights to. One approach to solving this problem may be to use a query-modification-like mechanism [11].

Quota enforcement is important because a single user could create numerous timer triggers, such that the total time required to process them is prohibitive. A simple approach to quota management might be to simply keep track of the CPU and I/O time being used by one user's triggers, and disable some or all of

those triggers if they exceed the users' daily quota. A more pro-active approach to quota enforcement may be possible whereby the timer trigger system can ask the DBMS how long queries for a particular user will take, and refuse to execute them if the total time is too large.

9 Conclusion

This paper has presented a clean, simple model for timer-driven triggers. The behavior of a timer-driven trigger is defined by a simple procedure. A system may implement timer-driven triggers with a different procedure, *as long as the triggers behave as if the original, simple procedure was used.* We challenge the database research community to develop designs and implementations of enhanced timer-driven trigger processing systems, within the parameters defined here.

References

[1] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker and J. Ullman, *The Asilomar Report on Database Research*, SIGMOD Record, 27 (1998), pp. 74-80.

[2] Categoric, *Categoric Alerts product literature, www.categoric.com*, 1999.

[3] R. Cochrane, H. Pirahesh and N. Mattos, *Integrating Triggers and Declarative Constraints in SQL Database Systems*, in T. M. Vijayaraman, ed., *Proceedings of the 22nd VLDB Conference*, 1996, pp. 567-578.

[4] Comshare, *Comshare Detect and Alert product literature, www.comshare.com*, 1999.

[5] C. J. Date and H. Darwen, *A Guide to the SQL Standard*, Addison-Wesley, 1993.

[6] A. Gupta and I. Mumick, eds., *Materialized Views: Techniques, Implementations and Applications*, MIT Press, 1999.

[7] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park and A. Vernon, *Scalable Trigger Processsing*, in M. Kitsuregawa, ed., *Proceedings of the IEEE Data Engineering Conference*, Sydney, Australia, 1999, pp. 266-275.

[8] Microsoft, *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*, Microsoft Press, 1997.

[9] NetMind, *NetMind Inc. home page, www.netmind.com*, 1999.

[10] L. X. Noronha, *Enhanced Techniques for Timer Trigger Processing*, MS thesis, CISE Department, University of Florida, Gainesville, FL, 1999.

[11] M. Stonebraker, *Implementation of Integrity Constraints and Views by Query Modification*, , *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1975, pp. 65-78.

[12] M. Stonebraker, *Inclusion of New Types in Relational Database Systems*, in M. Stonebraker, ed., *Readings in Database Systems*, Morgan Kaufmann, 1988, pp. 480-487.