TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS

Eric N. Hanson, Nabeel Al-Fayoumi, Chris Carnes, Mohktar Kandil,
Huisheng Liu, Ming Lu, J.B. Park and Albert Vernon
301 CSE
CISE Department
University of Florida
Gainesville, FL 32611-6120
(352) 392-2691
hanson@cise.ufl.edu
http://www.cise.ufl.edu/~hanson

TR-97-024
December 15, 1997

Abstract

This paper describes the architecture of the TriggerMan asynchronous trigger processor. TriggerMan processes triggers after updates have committed in the source database. It is designed to be able to gather updates from a wide variety of sources, including relational databases, object-relational databases, legacy databases, flat files, the web, and others. TriggerMan achieves the ability to gather updates from so many sources using an extensible data source mechanism. TriggerMan supports single-table (single data-source) triggers, sophisticated multiple-table (multiple-data-source) triggers, and temporal triggers. It maintains persistent copies of derived data (including materialized views and time series objects) to enable high-performance trigger condition testing. In order to take advantage of an existing database storage manager rather then create a new one, TriggerMan is being implemented as an extension to an object-relational DBMS. To do parallel trigger condition testing, TriggerMan makes use of an optimized discrimination network and a novel query modification mechanism, coupled with the native parallel query processing capability of the underlying DBMS.

1. Introduction

There has been a great deal of interest in active database systems over the last ten years. Many database vendors now include active database capability (triggers or active rules) in their products. Nevertheless, a problem exists with many commercial trigger systems as well as research efforts into development of database triggers. Most work on database triggers follows the event-condition-action (ECA) rule model. In addition, trigger conditions are normally checked and actions are normally run in the same transaction as the triggering update event. In other words, the so-called immediate binding mode is used. The main difficulty with this approach is that if there are more than a few triggers, or even if there is one trigger whose condition is expensive to check, then update response time can become too slow. A general principle for designing high-throughput transaction processing (TP) systems put forward by Jim Gray can be paraphrased as follows: avoid doing extra work that is synchronous with transaction commit [Gray93]. Running rules just before commit violates this principle.

Moreover, many advances have been made in active database research which have yet to show up in database products because of their implementation complexity, or because of the expense involved in testing sophisticated trigger conditions. For example, sophisticated discrimination networks have been developed for testing rule conditions [Hans96]. In addition,

techniques have been developed for processing temporal triggers (triggers whose conditions are based on time, e.g. an increase of 20% in one hour). Neither of these approaches has been tried in a commercial DBMS.

In this paper, we propose a new kind of system called an asynchronous trigger processor, or ATP. An ATP is a system that can process triggers asynchronously, after updates have committed in a source database, or have been completed in some other data source. Processing triggers asynchronously avoids slowing down update transactions with trigger processing logic. Moreover, since an ATP could be used with many different source DBMSs, the effort to develop the trigger processing code could be amortized over use with more applications. Really, an arbitrary application program can be used to transmit descriptions of database updates (update descriptors) to the ATP, and triggers can be processed on top of these update descriptors. The ability to process triggers based on updates from many different sources can help make it economically viable to implement sophisticated trigger processing code.

We are currently developing an ATP called TriggerMan as a vehicle for investigating issues related to asynchronous trigger processing. Part of the motivation for TriggerMan has been the surge in popularity of asynchronous replication features in commercial database systems. In actual practice, to achieve replication of data in a distributed DBMS, most database customers greatly prefer asynchronous replication to a synchronous replication policy based on distributed transactions using two-phase commit. The reason for this is that update availability and response time are both better with asynchronous replication. This was a motivating factor behind the choice to move to an external, asynchronous trigger processor, which also would avoid slowing down updates. Furthermore, as shall be explained later, the update capture mechanism built in to asynchronous replication systems can be used to send update descriptors to an ATP.

With respect to related research, many active database systems have been developed, including POSTGRES, HiPAC, Ariel, the Starburst rule system, A-RDL, Chimera, and others [Wido96]. In addition, there has been a notion of "de-coupled" rule condition/action binding mode for some time, as introduced in HiPAC [McAr89]. However, the implicit assumption regarding de-coupled rule condition evaluation and action execution was that the DBMS itself would still do the needed work. This paper outlines an alternative architecture that would off-load rule condition testing and action execution to a separate system.

TriggerMan needs to maintain some persistent state information. This includes materialized views that are part of a database discrimination network called a Gator network [Hans97b] as well as time series objects used for temporal trigger condition testing. TriggerMan is designed to be extensible in a number of ways, including the ability to add new data sources, new data types, and new temporal functions. TriggerMan is being implemented as an extension module (DataBlade) in Informix Dynamic Server with Universal Data Option, which was until recently called Informix Universal Server (Informix). Hereafter, we will call it simply Informix. Informix is a commercially available object-relational DBMS. It is expected that most, or all, relational DBMS vendors will migrate to an object-relational framework over the next few years. Hence, a system like TriggerMan could be made to work as an extension of a number of different DBMS products in the future. TriggerMan can use all the built-in data types and extended, user-defined data types that are available in the DBMS in which it is installed.

TriggerMan can act as either of the following, or both at the same time:

1. an external asynchronous trigger processor that can be used to add sophisticated, high-performance trigger processing capability to any type of data source, and

2. a sophisticated trigger extension module for the DBMS in which it is installed (the *host DBMS*, Informix in our implementation).

It can be seen that 1 enables 2 since the local DBMS in which TriggerMan is installed can be turned into a data source for TriggerMan. Placing simple triggers on tables in the local DBMS to capture updates can do this. These updates can then be passed asynchronously to TriggerMan. We are also examining ways to make it possible for TriggerMan to provide enhanced synchronous trigger processing capability for the host DBMS.

Other object-relational DBMS products have (or are expected to have) an architecture comparable to Informix. Hence, the techniques outlined in this paper for adding asynchronous trigger processing as an extension module could be used with these DBMSs as well.

In the rest of the paper, we describe the TriggerMan command language, system architecture, and trigger condition testing capability. The later is based on an advanced database discrimination network called Gator [Hans97b] and a novel query modification technique that allows re-use of the built-in parallel query processing capability in the host DBMS.

2. The TriggerMan Command Language

Commands in TriggerMan have a keyword-delimited, SQL-like syntax. TriggerMan supports the notion of a connection to a remote database or a generic data source program. A connection description for a remote database contains information about the host name where the database resides, the type of database system running (e.g. Informix, Oracle, Sybase, DB2 etc.), the name of the database server, a user ID, and a password. A single connection is designated as the default connection. There can be multiple data sources defined for a single connection. Data sources can be defined using this command:

```
define data source [connectionName.]sourceName [as localName]
[ ( attributeList ) ]
[ propertyName=propertyString,
...
  propertyName=propertyString ]
```

Suppose a connection “salesDB” had been defined on a remote database called “sales.” An example data source definition for the table “sale” in the sales database might look like this:

```
define data source salesDB.sale as sale
```

This command would read the schema from the salesDB connection for the “sale” table to gather the necessary information to allow triggers to be defined on that table.

Triggers can be defined using the following command:

create trigger <triggerName> [in *setName*] [-inactive]
from *fromList*
[on *eventSpec*]
[start time *timePoint*]
[end time *timePoint*]
[calendar *calendarName*]
[when *condition*]
[group by *attr-list*]
[having *group-condition*]
do *action*

Triggers are normally eligible to run as soon as they are created if a triggering event occurs. However, if the **-inactive** flag is specified the trigger remains ineligible to run until it is enabled later using a separate **activate trigger** command. The **start time** and **end time** clauses define an interval within which the trigger can be eligible to run. In addition, the name of a calendar object can be specified as part of a trigger. A calendar indicates "on" and "off" time periods. For example, a simple business calendar might specify "on" periods to be Monday through Friday from 8:00AM to 5:00PM. A trigger with a calendar is only eligible to be triggered during an "on" period for the calendar. In addition, a trigger is eligible to be triggered only if:

(1) the trigger is active, and

(2) the current time is between the start time and end time, and

(3) the calendar is in an "on" time period.

Triggers can be added to a specific trigger set. Otherwise they belong to a default trigger set. The **from**, **on**, and **when** clauses are normally present to specify the trigger condition. Optionally, **group by** and **having** clauses, similar to those available in SQL [Date93], can be used to specify trigger conditions involving aggregates or temporal functions. Multiple remote tables (or other data streams) can be referenced in the **from** clause. This allows multiple-table triggers to be defined.

An example of a rule, based on an **emp** table from a database for which a connection has been defined, is given below. This rule sets the salary of Fred to the salary of Bob:

create trigger updateFred
from emp
on update emp.salary
when emp.name = "Bob"
do execSQL "update emp set salary=:NEW.emp.salary where emp.name= 'Fred'"

This rule illustrates the use of an execSQL TriggerMan command that allows SQL statements to be run against data source databases. The :NEW notation in the rule action (the **do** clause) allows reference to new updated data values, the new emp.salary value in this case. Similarly, :OLD allows access to data values that were current just before an update. Values matching the trigger condition are substituted into the trigger action using macro substitution. After substitution, the trigger action is evaluated. This procedure binds the rule condition to the rule action.

An example of a more sophisticated rule (one whose condition involves joins) is as follows. Consider the following schema for part of a real-estate database, which would be imported by TriggerMan using **define data source** commands:

house(hno,address,price,nno,spno)
salesperson(spno,name,phone)
represents(spno,nno)
neighborhood(nno,name,location)

A rule on this schema might be "if a new house is added which is in a neighborhood that salesperson Iris represents then notify her," i.e.:

```
create trigger IrisHouseAlert
on insert to house
from salesperson s, house h, represents r
when s.name = 'Iris' and s.spno=r.spno and r.nno=h.nno
do raise event NewHouseInIrisNeighborhood(h.hno, h.address)
```

This command refers to three tables. The **raise event** command used in the rule action is a special command that allows rule actions to communicate with the outside world [Hans97]. Application programs written using a library provided with TriggerMan can register for events. When triggers raise events, the applications registered for the events will be notified. Applications can run on machines running anywhere on the network that is reachable from the machine where TriggerMan is running.

3. System Architecture

The TriggerMan architecture is made up of the following components:

1. the **TriggerMan DataBlade** which lives inside of Informix,
2. **data source applications**, which are programs that transmit a sequence of update descriptors to TriggerMan describing updates that have occurred in data sources,
3. **TriggerMan client applications**, which create triggers, drop triggers, register for events, receive event notifications when triggers fire, etc.,
4. one or more instances of the **TriggerMan driver** program, each of which periodically invokes a special TmanTest() function in the TriggerMan DataBlade, allowing trigger condition testing and action execution to be performed,
5. the **TriggerMan console**, a special application program that lets a user directly interact with the system to create triggers, drop triggers, start the system, shut it down, etc.

The general architecture of the TriggerMan system is illustrated in Figure 1. Having more than one TriggerMan driver program concurrently running the TmanTest() function can allow higher throughput for TriggerMan by letting more concurrent trigger processing activity take place inside the host DBMS server. Exactly how to choose the optimal multiprogramming level, including the number of TriggerMan drivers and the frequency with which they call TmanTest() is a topic for future research. Initially, the number of drivers and the frequency with which they call TmanTest() will be parameters that have reasonable defaults and can be set manually to other values. An initial choice for the defaults that may be appropriate is to have N drivers and a random interval between calls with mean N seconds, on a per-driver basis. N could initially be chosen to be equal to the number of processors, up to a limit of eight total drivers. We are also investigating the possibility of having data source applications call TmanTest() immediately after delivering update descriptors, in order to reduce trigger activation response time and avoid the need for polling.

Two libraries that come with TriggerMan allow writing of client applications and data source programs. These libraries define the TriggerMan *client application programming interface* (API) and the TriggerMan *data source API.* The console program and other application programs use client API functions to connect to TriggerMan, issue commands, register for events, and so forth. Data source programs can be written using the data source API. Examples of data source programs are a generic data source that sends a stream of update descriptors to TriggerMan, and a DBMS gateway program that gathers updates from a DBMS and sends them to TriggerMan.

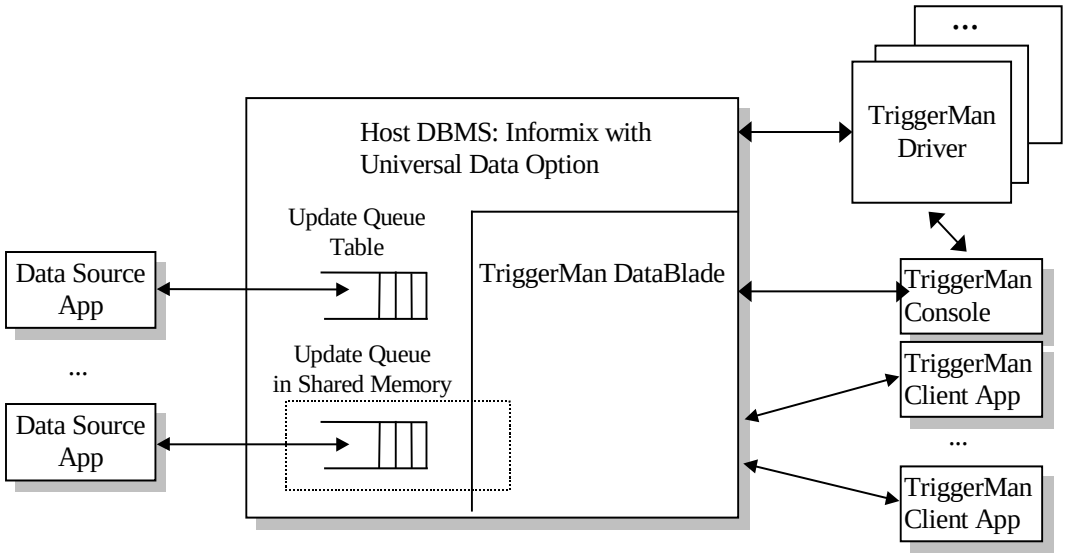


Figure 1. The architecture of the TriggerMan asynchronous trigger processor.

As Figure 1 shows, data source programs can either place update descriptors in a persistent queue, or in a volatile queue in shared memory. A persistent update queue is really just a special host DBMS table created by TriggerMan. A volatile queue in shared memory is used to hold update descriptors that do not need to be recovered in case of system failure. It is up to the application designer to determine if update descriptors sent to TriggerMan need to be recoverable.

4. Data Source Design

A flexible strategy is being designed to gather streams of update descriptors or other messages from data sources [Carn98]. A simple, generic data source could be an application that sends a stream of new data values to TriggerMan. Such a generic data source would be a program written using the data source API. A more sophisticated data source could gather a stream of update descriptors from a database by cooperating with the replication services provided by the DBMS. E.g. with Sybase, a gateway program could be written using the TriggerMan data source API and the Sybase replication API [Syba96]. This Gateway program

would transmit update descriptors received from the Sybase replication server and propagate them to TriggerMan. A different gateway program could be written for each potential DBMS that might serve as a data source. For databases for which no replication service exists, a gateway program could be written that would query the database periodically and compare the answers to the queries to produce update descriptors to send to TriggerMan [Lab96]. Alternatively, the gateway could trap inserts, updates and deletes using simple triggers in the source DBMS. Reading the database log is another alternative, but it is not usually realistic because DBMS vendors normally have a proprietary log format that other systems are not allowed to read, since the vendor reserves the right to change the log format.

TriggerMan maintains catalogs and other persistent state information using the capabilities of the host DBMS. To preserve transaction semantics, the first approach to handling the stream of updates from a DBMS will be to apply the updates in TriggerMan, and run the resulting trigger actions, in commit order. The Sybase replication server, for example, presents updates in commit order, making this strategy feasible. The maximum transaction ID handled so far by TriggerMan will be recorded along with updates to TriggerMan's internal state information in a single transaction. If this transaction fails, the updates will be rolled back and will be re-applied later. Maintaining the maximum transaction ID applied so far will make sure TriggerMan does not forget to handle a transaction from the primary database. Processing updates in commit order will require that only one TriggerMan driver process be responsible for all updates from a particular data source DBMS.

An issue that will be addressed in the future is how to deal with high update rates in the data source databases. If updates are taking place at a high rate in the source DBMS, TriggerMan might not be able to keep up with the source if it must handle the updates in commit order. This is because it might not be possible to get enough concurrency or parallelism in the ATP if the updates are handled serially. Possible solutions to this problem will be considered, such as relaxing the requirement to handle updates in commit order (for database data/data sources) or in the order of arrival (for generic data sources). Piggybacking multiple update descriptors in a single message to the TriggerMan server may also make it possible to handle higher update rates.

5. Support for Multiple-Table and Parallel Trigger Condition Testing

Allowing triggers to have conditions based on multiple tables greatly increases the power and expressiveness of the trigger language. However, efficiently testing multiple-table trigger conditions is a challenging problem. As part of prior work, we have developed an optimized multiple-table trigger condition testing mechanism known as the Gator network, a generalization of the TREAT and Rete networks used for rule condition testing in production rule systems such as OPS5 [Hans97b]. Here, we present a strategy that will allow parallel rule condition testing based on a Gator network to be done using a query modification strategy combined with the native parallel query processing capability of the host DBMS.

A different approach has been successfully used to perform parallel rule condition matching in main-memory production rule systems using a Rete network [Forg82] organized as a global distributed hash table, or GDHT [Acha92]. We considered using a similar technique but opted against it because it would be too difficult to implement. Using a GDHT approach, it would not have been possible to re-use the existing storage manager and query processor of the host DBMS to achieve fast processing of join triggers. A parallel storage manager and something comparable to a hash-based parallel query processing strategy would have to have been implemented from scratch. Using the approach of extending a host DBMS to create TriggerMan, the resulting

system should also be more robust and useable with real-world data sources for testing and evaluation purposes.

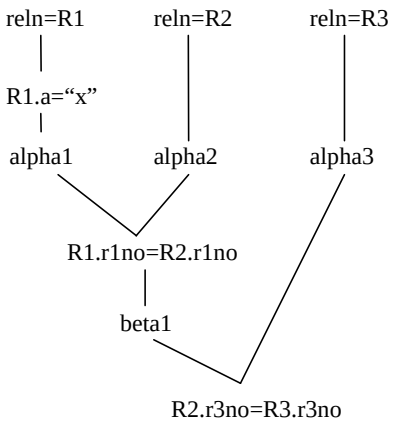
5.1. Overview of TriggerMan Trigger Processing Strategy

The idea behind the trigger processing strategy used in TriggerMan is as follows. A Gator network tree structure will be constructed for each trigger. Gator networks consist of nodes to test selection and join conditions, plus "memory" nodes that hold sets of tuples matching one or more selection and join conditions. Memory nodes that hold the result of applying a selection condition are called alpha nodes. Memory nodes that hold the result of the join of two or more other memory nodes are called beta nodes. Beta nodes can have two or more other memory nodes as inputs. In other words, the Gator network tree need not be a binary tree. It can be a bushy rooted tree, where the root is normally drawn at the bottom. In Rete network terminology, the root is called the P-node.

As an example, consider the following table schemas, trigger definition, and one possible Gator network for the trigger:

```
R1(r1no,a,b)
R2(r1no,r3no)
R3(r3no,c,d)

create trigger T1
from R1, R2, R3
when R1.r1no=R2.r1no
and R2.r3no=R3.r3no
and R1.a = "x"
then do ...
```



In this Gator network, memory alpha1 logically contains the result of

select * from R1 where R1.a= "x"

Similarly, beta1 logically contains the result of

select * from R1, R2 where R1.a = "x" and R1.r1no=R2.r1no

In addition, memory nodes can be either stored or virtual. A stored node is analogous to a materialized view. It actually contains the specified tuples. A virtual node is analogous to a real view. It only contains a predicate defining which tuples should qualify. It does not contain the real tuples. Only alpha memory nodes can be virtual.

A detailed discussion of how to choose the best shape of a Gator network, and how Gator networks can be used for multiple-table trigger condition testing on a single processor can be found in a separate paper [Hans97b]. A presentation of how the Gator network optimizer for TriggerMan will work is beyond the scope of this paper. However, techniques based on the work shown in our prior paper [Hans97b] will be used.

The Gator network trigger condition testing method presented below makes use of (1) extensible DBMS capability, (2) the query processor of the host DBMS (via the SQL interface), (3) the storage management capability of the host DBMS, and (4) a query modification procedure. Since access to the host DBMS capabilities is primarily through SQL, this trigger condition testing strategy should work on any type of hardware – single processor, shared-memory multiprocessor (SMP), shared-nothing, or a cluster of SMPs connected by a fast interconnect. The SQL interface to the DBMS will shield the trigger condition testing system from details of the underlying hardware. It will also allow parallelism to be exploited where possible.

Gator network rule condition testing is based on the notion of *tokens*. A token can be either a + token (one generated by an insert), or a – token (one generated by a delete). A token is a *simple token* if it corresponds to a single tuple from a data source. A token is a *compound token* if it is the join of one or more tuples with a newly inserted (deleted) token.

An outline of our approach to implementing Gator-network-based trigger condition testing is as follows:

1. The **from** and **when** clause part of every trigger condition is represented as a rule condition graph, or RCG. This graph has the same structure as a query graph. It contains one node for each tuple variable and an edge for each join condition. Single-tuple-variable selections are attached directly to the node to which they apply.
2. A selection predicate index (SPI) [Hans96b] is built from all the selection predicates of all the rules.
3. A Gator network tree structure is optimized for each rule. The tree structures may contain common sub-trees.
4. Stored Gator network memory nodes are maintained as tables in the host DBMS. Virtual alpha nodes are implemented as host DBMS views. TriggerMan automatically creates these table and view definitions.
5. Gator network memory nodes are "primed" at rule creation time by running queries to load them with data. The structure of these queries is determined from the rule condition.
6. When an update descriptor arrives it is packaged as a token. This token is passed through the SPI. If it matches a selection condition, it is passed on to the node underneath that condition.
7. If a + token arrives at a memory node, and that node has one or more siblings (nodes which feed into the same beta node or P-node) then *query modification* is performed, and a query is submitted to the host DBMS to find any tuples matching that token. This query will be run in parallel by the host DBMS query processor on a parallel machine if possible.[1]
8. If the query returns no results, stop. Otherwise, take the results, insert them into the node X immediately below the current node, and recursively invoke the rule condition testing process on X.
9. If the rule condition testing process is invoked on the P-node of a rule, fire the rule action for the data that just arrived at the P-node.

[1] This query modification process and reuse of the Informix query processor allows scaleable parallel join trigger condition testing.

A similar algorithm to that outlined in steps 6-8 also works for deletes; deletes will be ignored here. Updates can be handled as deletes followed by inserts. Below, an example is given to illustrate token flow and the query modification process. Then, a detailed description of the query modification algorithm and pattern-matching algorithm is given, elaborating on the outline above.

Consider the following trigger definition on data source tables S1, S2, S3 and S4.

```
create trigger T2
from S1, S2, S3, S4
when S1.x = S2.x and S2.y = S3.y and S3.z = S4.z
and S1.a = C1 and S4.b = C2
do ...
```

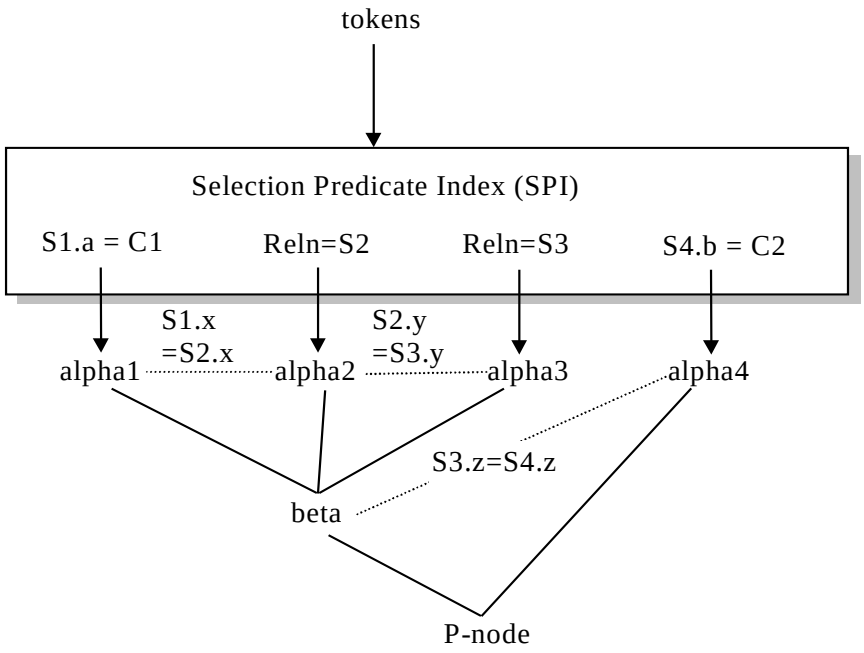


Figure 2. A Gator network for trigger T2.

Assume that all the attributes referenced in this trigger are legal attributes of the named tables. The symbols C1 and C2 represent constants. A possible Gator network for this trigger is given in Figure 2. There are join links between alpha1 and alpha2, alpha2 and alpha3, and beta and alpha4, as indicated by labels in the diagram. Suppose a tuple is inserted in S1, and it happens to have a value C1 in its "a" attribute. This tuple will be packaged as a + token, t, and passed to the rule condition testing system. Then it will match the condition S1.a=C1, and be inserted into alpha1. At that point, the system will perform query modification based on the structure of the Gator network and the rule condition. The following query, Q1, will be formed:

```
select t.*, alpha2.*, alpha3.*
from alpha2, alpha3
where t.x = alpha2.x and alpha2.y=alpha3.y
```

Here, t.* is expanded into a list of one or more constants (the fields of token t), and t.x represents a constant extracted from the x field of token t. This query is then passed to the query processor for execution. As an optimization, a plan for this query will be prepared in advance, and the values of t.* and t.x will be substituted into it as parameters at run time. This will avoid repeated query optimization.

Let the results of this query be the set RQ1. If RQ1 is empty, processing stops. Otherwise, RQ1 is first inserted into beta. Then, if RQ1 is small, the following will be done:

```
for each tuple y in RQ1 do
    let RQ2 be the result of
      select y.*, alpha4.*
      from alpha4
      where y.z = alpha4.z
    insert RQ2 into the P-node
end
```

If RQ1 is large, then the following will be done:

```
save RQ1 as a temporary table
let RQ2 be the result of
      select RQ1.*, alpha4.*
      from RQ1, alpha4
      where RQ1.z = alpha4.z
insert RQ2 into the P-node
```

Saving RQ1 as a temporary table can be done in parallel on a parallel machine if a SELECT INTO statement is used. Hence, in the case when RQ1 is large, the step to save RQ1 as a temporary table need not be a bottleneck. The decision about when a temporary table such as RQ1 is "small" will be based on cost-based optimization criteria. Details of this are left for future work.

5.2. General Description of Query Modification and Data Propagation Process

We now present a general description of the query modification and token propagation process. A rule condition is represented by a rule condition graph *RCG*. It is assumed that the graph *RCG* is connected. If it is not connected, one or more edges with "true" join conditions will be added arbitrarily to make it connected. Hence, the possibility of rule conditions involving cross products can be safely ignored. This is a reasonable choice because cross products are rare. In general, a Gator network *G* consists of a set of nodes N_1 ... N_m. Each node can be a leaf node (i.e. an alpha node) or an inner node (i.e. a P-node or a beta node). Each inner node N_i has two or more child nodes. The child nodes of N_i are its input nodes. Associated with each inner node N_i is the following information:

1. A join edge set *JE* showing the two-way join conditions that link together the child nodes of N_i. In other words, there is a set of one or more edges $JE = \{e_1, e_2, ... e_k\}$ consisting of pairs of nodes that are members of the set of children of N_i. Attached to each edge *e* is a join condition *JC(e)* that is a function solely of attributes of the nodes connected by *e*.

2. A **hyper-edge set** H_i containing zero or more hyper-edges $\{h_1, h_2, ... h_l\}$ associated with the children of node N_i. Each hyper-edge h in H_i is a set of three or more nodes that are children of N_i, along with a condition that is a function of the attributes of all of those nodes. For example, if N_1, N_2 and N_3 are sibling nodes with parent node N_4, and there is a condition derived from the rule condition graph of the form $N_1.x + N_2.y + N_3.z > 10$, then a hyper-edge would be created for N_1, N_2 and N_3 using that condition. This hyper-edge would be attached to N_4. Normally, the hyper-edge set will be empty since conditions that are a function of more than two tuple variables at once are rare.

3. An **expensive selection predicate set** $ES_i = \{es_1, ... es_m\}$ to be tested against compound tokens before they can be inserted into N_i. These predicates may involve expensive functions, e.g. "matches(fingerprint,fingerprint)" might be an expensive function that could take seconds to compute for a pair of fingerprints.

An illustration of an example node N_i is given in Figure 3. This node has four child nodes. There are three join edges, e_1, e_2 and e_3, connecting the child nodes. In addition, there is a single expensive selection predicate es_1, and a single hyper-edge h_1 attached directly to N_i. Again, conditions involving three or more tuple variables (hyper-edges) and expensive selection conditions are both expected to be rare, so this is a fairly general example.

Normally, selection predicates are tested in the selection predicate index before all the join conditions. However, expensive selection predicates will be tested after applying one or more join operations. Our optimization strategy for placing expensive predicates in a Gator network is based on a cost-driven search strategy comparable to that developed for the POSTGRES object-relational DBMS [Ston90,Hell98], with some extensions. The optimization algorithm used is a subject of another paper [Kand98]. The focus here is on rule condition testing given a specific Gator network, not on finding the best Gator network.

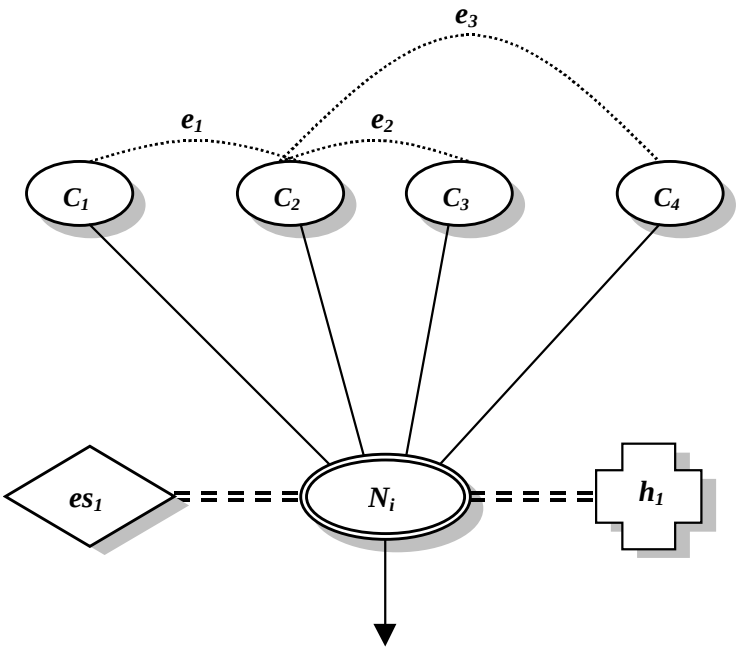


Figure 3 Example Gator network node, showing edges connecting child nodes, as well as expensive selection predicates and hyper-edges.

Consider the case of a set of one or more + tokens arriving at a node N in a Gator network G. There are two cases to consider: (1) the set of plus tokens is small, and (2) the set of + tokens is large. The crossover point between large and small will be based on cost-based optimization criteria. The details of this are left for future work. In case (1), tokens will be tested one at a time to see if any new tokens need to be passed to the parent node. In case (2), the entire set of tokens will be saved in a temporary relation TR, and tested together using a single query.

For each pair consisting of a child node N_i and its parent node P, there will be a **tuple query template** $TQT(N_i,P)$ and a **set query template** $SQT(N_i,P)$. These query templates are saved as part of the Gator network. The structure of a tuple query template is as follows, where symbol *s'* is equivalent to symbol *s* with all attributes whose source is N_i replaced by constants drawn from the corresponding attributes of token *t*:

select $N_1.*, N_2.*, \ldots, N_{i-1}.*, t.*, N_{i+1}, \ldots N_n.*$
from $N_1, N_2, \ldots, N_{i-1}, N_{i+1}, N_n$
where e_1' and e_2' … and e_k'
and h_1' and h_2' ... and h_l'
and es_1' and es_2' … and es_m'

In the target list of the above query, *t.** is expanded into a comma-separated list of constants extracted from the attributes of *t.*

The structure of a set query template $SQT(N_i,P)$ is as follows, where a symbol *s'* is equivalent to the symbol *s* with all attributes of N_i replaced by the attribute of a relation TEMP having the same name.

select $N_1.*, N_2.*, \ldots, N_{i-1}.*, TEMP.*, N_{i+1}, \ldots N_n.*$
from $N_1, N_2, \ldots, N_{i-1}, TEMP, N_{i+1}, N_n$
where e_1' and e_2' … and e_k'
and h_1' and h_2' ... and h_l'
and es_1' and es_2' … and es_m'

The goal when one or more tokens arrive at N_i is to determine whether any new tokens need to be inserted into the parent of N. Let TS be the name of the token set that just arrived at N_i. The following algorithm is used to continue propagating tokens through the network:[2]

```
Propagate(N, P,TS)
inputs: memory node N and its parent P, plus token set TS just added to N
outputs: none
actions: propagates tokens forward in the Gator network as needed
begin
  if N is a P-node then
    trigger the rule for P, using the tokens in TS
  else if TS is small then
    PropagateSmall(N, P, TS)
  else
    PropagateLarge(N, P, TS)
  end if
end
```

[2] To simplify presentation, the algorithm is given without considering common sub-expressions. If common sub-expressions are shared, each node could have more than one parent.

```
PropagateSmall(N, P, TS)
remarks: handle case (1)
begin
  Initialize TS2 to an empty set of tuples in main memory.
  for each token t in TS do
    Let Q be the result of substituting t into TQT(N,P).
    Run Q, appending the result to TS2.  If TS2 becomes large, convert it to
        a stored relation "on the fly."
  end
  Propagate(P, Parent(P), TS2)
end

PropagateLarge(N, P, TS)
remarks: handle case (2)
begin
  Let TEMP be the relation that holds TS.
  Let Q be the result of substituting TEMP into SQT(N,P).
  Initialize TS2 to an empty set of tuples in main memory.
  Run Q, appending the result to TS2.  If TS2 becomes large, convert it to
        a stored relation "on the fly."
  Propagate(P, Parent(P), TS2)
end
```

Temporary tuple sets and relations generated during execution of the above algorithm would be garbage-collected after the algorithm completes. The above algorithm has a number of important properties:

1. It uses the existing database query processor to do pattern matching across multiple input nodes of a beta node or P-node. Since the database query processor is highly optimized, this is crucial to get high performance. Another tremendous benefit of using the query processor is that it greatly reduces the implementation complexity of TriggerMan. The query processor can choose the best join order and access methods, run queries in parallel, and defer expensive predicate testing to the ideal time. Using the algorithm given above, TriggerMan can take advantage of all this.
2. It keeps the total number of queries run relatively small by dynamically converting from token-at-a-time propagation to set-at-a-time propagation when the set of tokens passed into a memory node is large. This is important since there is a per-query overhead for running a query.
3. It runs queries with fewer joins (derived from a tuple query template instead of a set query template) when there are only a few tokens that have just arrived at a memory node. Avoiding joins should result in lower costs in this situation.

These properties will help provide high-performance processing of triggers involving joins and expensive functions in TriggerMan, while keeping implementation complexity tractable. We now turn to a discussion of how to support triggers based on changes to data over time.

6. Temporal Trigger Support

Temporal triggers are triggers whose conditions are based on changes in a value or set of values over time. For example, a temporal trigger could be defined to fire if the sales from a particular store rise by more than 20% in one month. Prior work on temporal triggers has focused on logic-based trigger languages [Sist95]. The difficulty with these languages is that the user must specify the trigger in a logic-based notation, and logic-based languages with quantifiers may be difficult for typical application developers to master. Moreover, certain kinds of temporal conditions may be quite useful, yet be extremely difficult or impossible to specify using temporal logic. For example, one might envision a temporal trigger that would fire if the price of a stock had a "spike" in value, where the definition of "spike" is based on some application-specific mathematical criterion. This might be a criterion such as "the average root mean square difference on a point-by-point basis between the actual sequence of values (curve) and an ideal spike is less than a threshold value." An example of an ideal spike and what an observed spike might look like is given below:

ideal spike observed spike

The capability to detect a spike based on mathematical criteria would be much easier to express using an algorithmic language like C, C++, Java, or FORTRAN than using temporal logic. Moreover, temporal functions written in C, for example, may be able to evaluate temporal trigger conditions much more efficiently than the equivalent temporal logic-based condition evaluator.

Rather than use a temporal logic-based language, we propose to use a set of basic temporal functions, including **increase**, **decrease** and several others, as well as temporal aggregates such as the **sum** and **count** of values over a certain time window. The benefit of using temporal operators to specify trigger conditions is that they are declarative, and relatively simple to understand – you say what you want, not how to achieve it. The implementation of the basic temporal functions will be provided as a standard part of the system. In addition, a *temporal function extensibility mechanism* is being developed to allow sophisticated application developers to write code to implement new temporal functions and register this code with the TriggerMan system. The extension code will be dynamically linked by the TriggerMan system when needed.

As mentioned earlier, the TriggerMan trigger language supports temporal condition specification through the use of the **group by** and **having** clauses familiar to users of SQL. For example, the following trigger will fire when there is an increase or decrease of more than 20% in the price of IBM stock in a six month period.

```
create trigger BigIBMchange
from stock
when stock.symbol = “IBM”
having increase(stock.price, “20%”, “6 mo”) or
  decrease(stock.price, “20%”, “6 mo”)
do raise event BigChange (“IBM”)
```

The above trigger can be generalized to all “technology” stocks by introducing a **group by** clause and modifying the **when** clause, as follows:

```
create trigger BigTechStockChange
from stock
when stock.category = “technology”
group by stock.symbol
having increase(stock.price, “20%”, “6 mo”) or
  decrease(stock.price, “20%”, “6 mo”)
do raise event BigChange (stock.symbol)
```

The **group by** capability is powerful since it allows triggers for multiple groups to be defined using a single statement.

Temporal functions can return boolean values (temporal predicates) and scalar values, such as integers and floating point numbers. These types of temporal functions can be composed in the **having** clause to form compound temporal conditions. For example, the following temporal function might compute a moving average of a value over a time window of width window_size:

```
moving_avg (expr,window_size)
```

The following example shows how this function could be used in conjunction with the **increase** function to detect when the 10-day moving average of the price of General Motors stock increases by more than 15% in three months:

```
create trigger GM_TREND
from stock
when stock.symbol = “GM”
having increase(moving_avg(stock.price, “10 days”), “15%”, “3 months”)
do ...
```

Values of temporal aggregates computed in the **having** clause will sometimes need to be used in the trigger action. The trigger language needs a way to support this form of condition/action binding. When it is not ambiguous, the name of the temporal aggregate function or group by attribute can be used in the trigger action, preceded by a : symbol. This will be macro-expanded when the action is executed. The following example illustrates condition-action binding.

```
create trigger HighYearlySales
from sale
group by sale.spno
having sum(sale.amount, “1 yr”) > 1000000
do execSQL “append to highSales(:sale.spno, :sum, Date())”
```

If the same function appears multiple times, the **as** operator can be used to bind names to the values produced by those functions, e.g.:

```
create trigger HighSalesAndCommssions
from sale
group by sale.spno
having sum(sale.amount, "1 yr") as s1 > 500000
and sum(sale.commission, "1 yr") as s2 > 50000
do execSQL "append to highSales(:sale.spno, :s1, :s2, Date())"
```

A full description of the temporal trigger mechanism implementation plan for TriggerMan is beyond the scope of this paper, and will be treated more fully elsewhere [AlFa98]. However, the basic strategy is to automatically maintain data value histories as needed to test temporal conditions. These histories will be represented as *time series* objects. These time series objects will be instances of the time series data type provided by the Informix time series DataBlade, which is commercially available. Time series will be created implicitly and maintained incrementally, based on the structure of temporal trigger conditions. These time series will be used as needed to test temporal trigger conditions when possible triggering events occur, such as the insertion of a new value in a time series, or the expiration of a timer. For example, consider a trigger containing an **increase** operator. This trigger would have a time series created for it. Upon insertion of a new value in the time series, a function would be called to iterate over the time series to test the **increase** condition. It is possible to add new functions to Informix that make use of the internal structure of the time series. We will use this mechanism to add the temporal operators such as **increase**, **decrease**, **stayInRange**, etc. to TriggerMan. Sophisticated users will be able to develop their own new temporal operators as DLL functions, and register them with TriggerMan.

7. Type and Function Extensibility

TriggerMan is designed to be extensible. All built-in and extended data types supported by the host DBMS will be accessible to TriggerMan. This is highly desirable, since it will allow TriggerMan to function with a full set of advanced data types. There is a clear trend toward inclusion of type extensibility as a standard feature in mainstream commercial DBMS products. Active database tools need to be able to work with the wide variety of data types that future application developers will require. By directly using host DBMS extended data types, TriggerMan and systems like it can piggyback on the growing collection of vendors marketing extended data types for objected-relational DBMSs.

TriggerMan will open the host DBMS data type and function catalogs to inquire as needed about the data types and functions (methods) defined in the system. An expression evaluator will be built inside the TriggerMan DataBlade. In the selection predicate index, TriggerMan must perform its own expression evaluation. Those selection condition expressions placed in the TriggerMan SPI will be compiled into pseudo-code, which will be interpreted. Operators defined on extended data types will be evaluated by the TriggerMan expression interpreter when needed by obtaining pointers to functions implementing the operators, and then calling the functions via the pointers. These function pointers can be obtained using functions available for this purpose in the host DBMS type extension API (e.g. Informix provides such functions, which we will use in our implementation).

8. Alternative, Generic Architecture

An alternative to the TriggerMan architecture outlined in Figure 1 would be to implement an asynchronous trigger processor as a system that is purely an external database application, not an extension module in an extensible DBMS [Carn98]. A possible architecture for an external

version of an asynchronous trigger processor is shown in Figure 4. We decided not to pursue this approach in our prototype implementation for the following reasons:

1. It would require extra data transfer between the DBMS and TriggerMan, which would hurt performance. Both system boot-up time and token propagation time would be worsened.
2. It would not allow the use of the time series and calendar data types in the Informix time series DataBlade, which would greatly increase the effort required to implement temporal triggers. It would also require time series to be placed in an application address space rather than the DBMS, requiring much expensive data transfer between the DBMS and the application. In addition, complex decisions about what time series to cache in the application address space would have to be made.
3. It would make it difficult to use extended data types.

However, an external asynchronous trigger processor still could be made to work with reasonable performance using Gator networks and a query-modification-based approach to token propagation, as outlined in this paper. Catalogs, alpha nodes, and beta nodes would all be stored in the DBMS. A catalog cache, a Gator network skeleton, and the SPI would be stored in the memory of the external trigger processor application. In general it would not be necessary for a large amount of data to be stored in the external trigger processor's address space. The exception to this is that it would be necessary to cache time series objects for fast temporal trigger processing. No query processor, buffer manager, or transaction processing system would be needed in the external trigger processor. Care could be taken during token propagation to make sure that traffic between the DBMS and the external trigger processing application was kept to a minimum.

Such a system could be written using ODBC, a generic database interface, and could then use a wide variety of different database systems to store state information, including memory

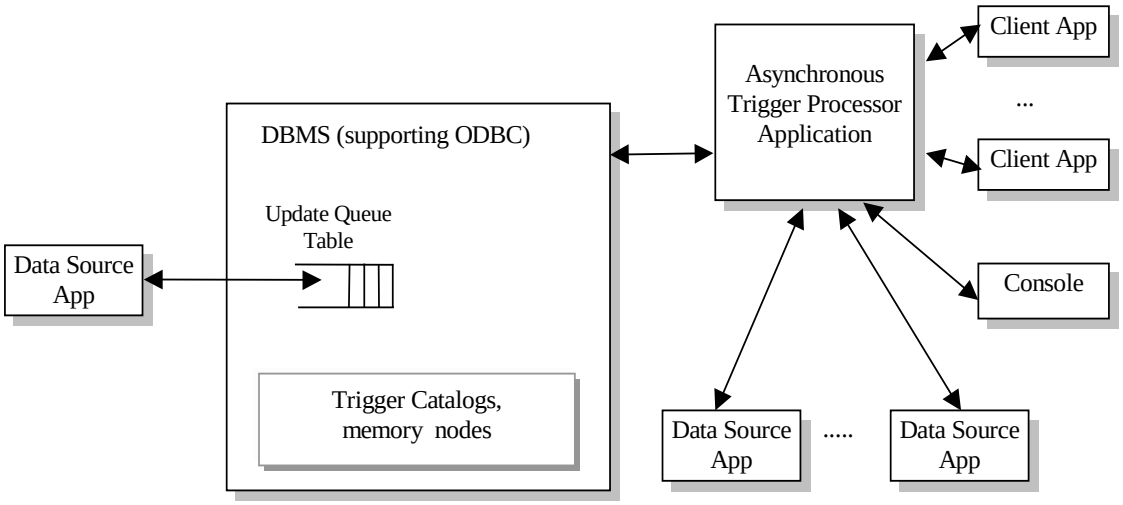


Figure 4. Possible external architecture for an asynchronous trigger processor.

nodes and catalogs. It would then not be tightly coupled to one DBMS. Moreover, it would automatically work with single-processor DBMSs, shared-memory parallel DBMSs, and massively-parallel shared-nothing DBMSs. Such a system would also be easier to install and administer. Though we are not directly exploring this approach, many of the results of the research outlined in this paper will be applicable to development of an external asynchronous trigger processor that functions in a similar way.

9. Conclusion

The research outlined here seeks to develop principles that will allow the effective construction of asynchronous, or "outboard" trigger processing systems. A prototype ATP called TriggerMan is currently being implemented as an extension of Informix as a vehicle to explore asynchronous trigger processing issues and to validate the design approach introduced here. Previously, we implemented a simple, stand-alone version of TriggerMan [Hans97c], and a significant portion of the code, including the entire parser, is being re-used in the Informix DataBlade version.

TriggerMan, or a system like it, could be useful in situations where current trigger systems are not. For example, TriggerMan could trigger on a stream of updates generated by a general application program that were never placed in any DBMS. Moreover, TriggerMan could place a trigger on two different data sources, one from a DBMS, and one from a program, performing an information fusion function. This type of function could be valuable in a number of heterogeneous information systems applications. For example, in a chemical plant application, a trigger could correlate a stream of reactor vessel temperature and pressure values sent by an application with known dangerous combinations of temperature and pressure kept in a database, firing when it saw a dangerous combination. The main benefit of an ATP system is that it can allow sophisticated, "expensive" triggers (i.e. multiple-table triggers, temporal triggers and triggers involving expensive functions) to be defined against a database and processed using the best available algorithms, without adversely impacting on-line update processing. This could greatly expand the benefits of trigger technology in demanding, update-intensive environments.

The results of the TriggerMan project could lead to a new type of system to support applications that need to monitor changes to information – an asynchronous trigger processor. In addition, an architecture for an asynchronous trigger processor similar to the one described here could be incorporated directly into a DBMS. In summary, the work outlined here can help develop a new, useful kind of information processing tool, the ATP, and point the way to improvements in the active database capability of existing database management systems. In either case, it will become possible to develop powerful and efficient information monitoring applications more easily.

Bibliography

[Acha92] Acharya, A., M. Tambe, and A. Gupta, "Implementation of Production Systems on Message-Passing Computers," IEEE Transactions on Knowledge and Data Engineering, 3(4), July 1992.

[AlFa98] Al-Fayoumi, Nabeel, *Temporal Trigger Processing in the TriggerMan Active DBMS*, Ph.D. dissertation, Univ. of Florida, 1998. In preparation.

[Arno96] Arnold, K., J. Gosling, *The Java Programming Language*, Addison Wesley Longman, 1996.

[Carn98] Carnes, Chris, *A Flexible Data Source Architecture for an Asynchronous Trigger Processor, Ph.D. Dissertation,* University of Florida. In preparation. 1998.

[Date93] Date, C. J. And Hugh Darwen, *A Guide to the SQL Standard*, 3rd Edition, Addison Wesley, 1993.

[Forg82] Forgy, C. L., Rete: "A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

[Gray93] Gray, Jim, *Transaction Processing, Concepts and Techniques*, Morgan Kaufmann, 1993.

[Hans90] Hanson, Eric N., M. Chaabouni*, C. Kim and Y. Wang*, "A Predicate Matching Algorithm for Database Rule Systems," *Proceedings of the ACM-SIGMOD Conference on Management of Data, pp. 27*1-280, Atlantic City, NJ, June 1990.

[Hans96] Hanson, Eric N., "The Design and Implementation of the Ariel Active Database Rule System," *IEEE Transactions on. Knowledge and Data Engineering*, vol. 8, no. 1, pp. 157-172, Feb., 1996.

[Hans96b] Hanson, Eric N. and Theodore Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," *Information Systems*, vol. 21, no. 3, pp. 269-298, 1996.

[Hans95] Hanson, Eric N., S. Bodagala, M. Hasan, G. Kulkarni, J. Rangarajan, *Optimized Rule Condition Testing in Ariel Using Gator Networks*, University of Florida CISE Department TR 95-027, http://www.cise.ufl.edu, October 1995.

[Hans97] Hanson, Eric N. et al., "Flexible and Recoverable Interaction Between Applications and Active Databases," *VLDB Journal*, 1997 (accepted).

[Hans97b] Hanson, Eric N., Sreenath Bodagala, and Ullas Chadaga, "Optimized Trigger Condition Testing in Ariel using Gator Networks," University of Florida CISE Dept. Tech. Report 97-021, November 1997. http://www.cise.ufl.edu.

[Hans97c] Hanson, Eric N. and Samir Khosla, "An Introduction to the TriggerMan Asynchronous Trigger Processor," *Proceedings of the 3rd Intl. Workshop on Rules In Database Systems,* Skovde, Sweden, June 1997.

[Hell98] Hellerstein, J., "Optimization Techniques for Queries with Expensive Methods**,"** to appear, *ACM Transactions on Database Systems (TODS),* 1998. Available at www.cs.berkeley.edu/~jmh.

[Info97] "Informix Dynamic Server, Universal Data Option," http://www.informix.com.

[Kand98] Kandil, Mohktar, *Predicate Placement in Active Database Discrimination Networks*, PhD Dissertation, Univ. of Florida, 1998. In preparation.

[Lab96] Labio, Wilbert J. and Hector Garcia-Molina, Efficient Snapshot Differential Algorithms for Data Warehousing, Proceedings of the 1996 VLDB Conference, Sept., 1996, pp. 63-74.

[McCa89] "McCarthy, Dennis R. and Umeshwar Dayal, "The Architecture of an Active Data Base Management System," *Proceedings of the. ACM SIGMOD Conference on Management of Data.*, Portland, OR, June, 1989, pp. 215-224.

[Sist95] Sistla, Prasad A. and Ouri Wolfson, "Temporal Triggers in Active Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 3, June, 1995, pp. 471-486.

[Ston90] Stonebraker, Michael., Larry Rowe and Michael Hirohama, "The Implementation

of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 7, March, 1990, pp. 125-142.

[Syba96] Sybase Replication Server Technical Overview, Sybase Inc., 1996.

[Wido96] Widom, J. And S. Ceri, *Active Database Systems*, Morgan Kaufmann, 1996.

[Witk93] Witkowski, A., F. Carino and P. Kostamaa, "NCR 3700 – The Next-Generation Industrial Database Computer," *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.