An Introduction to the TriggerMan Asynchronous Trigger Processor[†]

Eric N. Hanson and Samir Khosla[‡]
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-97-007

April, 1997

Abstract

A new type of system for testing trigger conditions and running trigger actions outside of a DBMS is proposed in this paper. Such a system is called an *asynchronous trigger processor* since it processes triggers asynchronously, after triggering updates have committed in the source database. The architecture of a prototype asynchronous trigger processor called TriggerMan is described. TriggerMan 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 can make use of the asynchronous replication features of commercial database products such as Sybase to gather updates. When cooperating with a source DBMS with direct support for asynchronous replication, TriggerMan can gather updates in an efficient and robust manner. TriggerMan supports simple, single-table (single data-source) triggers, as well as sophisticated multi-table (multi-data-source) triggers. It also will support temporal triggers using an extensible temporal function mechanism.

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) 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

[†] This work was supported by the United States Air Force Rome Labs and NCR/Teradata.
[‡] Currently with Informix Software Corporation, samir@informix.com.

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 product.

In this paper, the author proposes 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 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. A simple subset of the functionality discussed in this paper has been implemented. We are actively doing the detailed design and implementation of the more advanced features of TriggerMan.

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, there has been a notion of “de-coupled” rule condition/action binding mode for some time, as introduced in the HiPAC active DBMS project [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.

Ultimately, the proposed TriggerMan/ATP architecture will provide an “active information server” capability that can support a wide variety of applications. This architecture will be able to support different tasks that involve monitoring of information sources, filtering of data, and selective propagation of information. TriggerMan can be used to augment traditional data management applications, as well as support new, distributed, heterogeneous information systems applications.

TriggerMan will be extensible in a number of ways, including the ability to add new data sources, new data types, and new temporal functions. To handle extended data types, the approach used will be similar to that used in object-relational database systems such as Informix Universal server. Either an existing standard such as the Informix DataBlade format will be adopted, or a comparable facility will be developed.

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, etc.), the name of the database server, a userid, 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] [(
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> [-inactive] [in triggerset]
from fromList
[on eventSpec]
[when condition]
[group by attr-list]
[having group-condition]
do action
```

Triggers are implicitly active (eligible to run) as soon as they are created, unless the -inactive flag is specified. Triggers can be added to specific a 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 object 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(:NEW.h.hno, :NEW.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.

System Architecture

The general architecture of the TriggerMan system is illustrated in Figure 1. Each box in this diagram represents a process. These processes can run on the same machine or different machines. The TriggerMan process has a parallel internal structure, consisting of a number of virtual processors, or *vprocs*. The vprocs communicate with each other via message passing. Hence, the TriggerMan server code can be made to run with little modification on shared-memory multiprocessors, shared-nothing machines, and collections of SMP machines connected by an interconnect. The first parallel implementation is designed to run on an SMP. The vproc concept has been used before successfully in the implementation of parallel DBMS software, such as the Teradata system [Witk93].

In the current TriggerMan system, each vproc contains multiple threads, including:

- a *matching thread* that processes update descriptors arriving from data sources to see if trigger conditions are satisfied,
- a *command server thread* that handles requests from client applications, and
- *rule action execution threads* to run rule actions.

The number of vprocs is normally made equal to the number of real processors in the system. In the current implementation, single-table triggers are allocated to different vprocs in a round-robin fashion to allow parallel condition testing. A special client application called the Console allows a user to start the system, shut down the system, create triggers, define data sources, and run other commands supported by TriggerMan. Multiple threads, spread across the vprocs, allow parallelism on an SMP machine.

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, such as a generic data source that sends a stream of update descriptors to TriggerMan, or a DBMS gateway program that gathers updates from a DBMS and sends them to TriggerMan, can be written using the data source API.

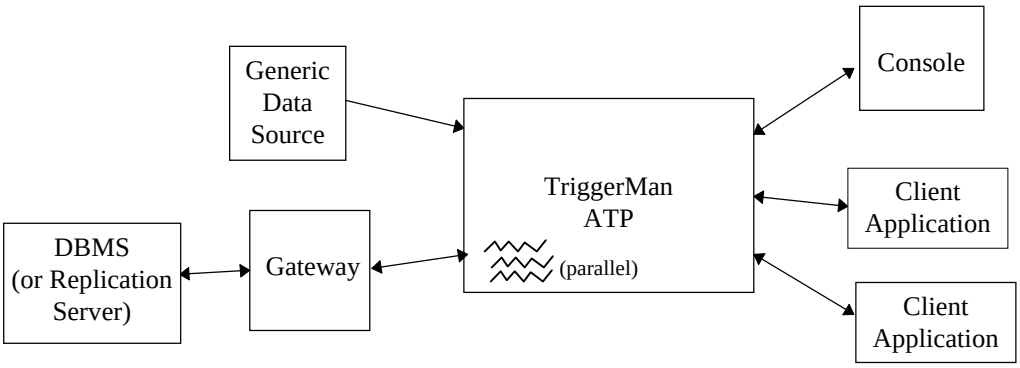

Figure 1 Architecture of the TriggerMan asynchronous trigger processor.

Data Source Design

A flexible strategy is being designed to gather streams of update descriptors or other messages from data sources. A simple, generic data source could be an application that sends a stream of new data values to TriggerMan. Such a generic data source, as illustrated in Figure 1, 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, as shown in Figure 1, 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 [Chaw96]. 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 will maintain catalogs and other persistent state information using a transactional 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.

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).

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 criteria, 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** and **decrease**, as well as temporal aggregates such as the **sum** and **count** of values over a certain time window. The implementation of these basic temporal functions will be done using C and C++ code. In addition, a *temporal function extensibility mechanism* is being developed to allow sophisticated application developers to write code in C and C++ to implement new temporal functions and register this code with the TriggerMan system. The extension code will be dynamically linked by the TriggerMan server 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 (:NEW.stock.symbol)
```

This temporal trigger capability can support temporal functions that 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 Oracle stock increases by more than 15%:

```
create trigger ORACLE_TREND
from stock
when stock.symbol = “ORCL”
having increase(moving_avg(stock.price, “10 days”), “15%”)
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 can be used in the trigger action, as in the following example:

```
create trigger HighYearlySales
from sale
group by sale.spno
having sum(sale.amount, “1 yr”) > 1000000
do execSQL “append to highSales(:NEW.sale.spno, :NEW.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 > 10000
do execSQL “append to highSales(:NEW.sale.spno, :NEW.s1, :NEW.s2, Date())”
```

Adding New Temporal Functions

The temporal function mechanism in TriggerMan is designed to be extensible. A new temporal operator can be defined using this notation:

```
define temporal function returnType funcName (argumentDefinition)
    dynamicLinkLibraryName functionPrefix
```

The dynamicLinkLibraryName is the name of the dynamic link library (DLL) where relevant functions are kept. The functionPrefix is the prefix of the name of all functions that are relevant. A temporal function’s argument list can

specify normal arguments, as well as initialization arguments used to initialize the state of the temporal function. Initialization arguments are preceded by the keyword **init**. A default value can also be provided for **init** arguments. For example, a new function to compute an exponential average could be registered with the system like this (the /local/db/expavg.so file is a shared object file that can be dynamically linked):

```
define temporal function double expavg (double newValue, init double multiplier = 0.9)
  "/local/db/expavg.so" "double_expavg"
```

Also, functions can be overloaded. For example, expavg can be re-implemented for different types, such as float, and the system will automatically use the right function depending on the data types with which it is called.

Functions with the following formats and naming conventions need to be available in a dynamic link library to implement a temporal function. This kind of extensibility technique is similar to that used in extensible database systems such as POSTGRES [Ston90] and the Informix Universal Server [Info97]:

function format	description
void* funcPrefix_new(int numInitArgs, void **initArgs);	Constructor to make temporal function state information object. Must take an argument count and an array of arguments of type void*. These arguments are used to pass zero or more initialization values for this temporal function. The function must not assume that storage for initialization arguments will not be reclaimed.
void funcPrefix_delete(void *state)	Destructor function to delete state information.
int funcPrefix_includeValue(void *state, void *newValue)	Function to adjust state to include a new value in a sequence of values.
int funcPrefix_resetState(void *state)	Reset state information.
int funcPrefix_dropValue(void *state)	Function to drop oldest value in sequence.
int funcPrefix_currentValue(void *state, void **output)	Function to get current value of function. Sets **output to point to result.
int funcPrefix_getPrintValue(void *state, char *printBuf),	Place a printable (character string) representation of the current value of the function in printBuf.

The C language is used rather than C++ to define this interface to simplify the dynamic linking process, and to ensure compatibility with legacy C code that might be needed to define temporal operators. If necessary, a small set of "wrapper" C functions could be used to define the interface to a C++ class used to maintain state for a temporal function. A Java [Arno96] version of this extensibility mechanism is also be considered.

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 [Hans96c]. We are investigating strategies that will allow development of a parallel version of the Gator network, as well as cost models and caching strategies specific to the environment of an asynchronous trigger processor. These new cost models and caching strategies are needed since TriggerMan runs in an separate address space and possibly on a separate machine from the DBMS or other data sources.

The current design plan is to use a *global, distributed hash table* (GDHT) approach to implement a parallel, multiple-table rule condition testing mechanism in a parallel version of Gator networks. The GDHT approach to pattern matching for multiple-table triggers involves hash partitioning the state information necessary to do rule condition testing across all the processors in a parallel machine. In the case of TriggerMan, this information would be split across the set of vprocs. By partitioning this information, the pattern matching needed to test multiple-table trigger conditions can be carried out in parallel. The full details of the GDHT approach are beyond the scope of this

paper. However, this approach has been successfully used to perform parallel rule condition matching in main-memory production systems using a Rete network [Forg82] organized as a GDHT [Acha92].

Extensibility

TriggerMan is designed to be extensible. This will include support for stored procedures written in Tcl, new data types and operators, and stored procedures written in C, C++, and perhaps Java, in addition to new temporal functions. Support for extended data types such as images, time series, web pages, text, etc. within a database management system has been supported in POSTGRES [Ston90] and Informix [Info97] and is being included in other commercial database products. This feature is extremely popular and is beginning to be used to add multimedia and object management capability to real-world database applications. The approach taken to support extensibility has been to define a library of C functions with a certain format, including (1) constructor and destructor functions, (2) functions for translating an instance of a type from internal to external format, (3) functions for performing operations on instances of a type, (4) functions for estimating the cost of performing certain operations, etc. This library is then registered with the DBMS (similar to the paper given earlier for supporting temporal functions). As an example, an extended data type called Document could be created, and triggers could be defined on a stream of documents arriving from an intelligence-gathering source, implementing a form of selective dissemination of information using TriggerMan.

If possible, the approach to handling extended data types in TriggerMan will be to use the standard extensibility format used by Informix, and introduce commands to register new types with the system. If this is done, the same DataBlade modules used by Informix can be used by TriggerMan. Using the existing Informix type extension standard in TriggerMan is preferred to defining a new standard for extended types, since existing extended types that have already been implemented for Informix could be used with TriggerMan. As part of the TriggerMan project, issues related to moving large objects between a DBMS and TriggerMan, caching the internal representation of large objects, and evaluating expensive predicates within the TriggerMan server will be investigated.

Performance of Initial Prototype

A version of the prototype TriggerMan server is already operational which implements single-table triggers, but has no persistent catalogs or ability to directly communicate with a DBMS via a replication server gateway. It supports parallelism using the concept of virtual processors (vprocs), making the code portable to both SMP and shared-nothing parallel computers. Performance tests were run on a dual-processor 75Mhz Sun SPARCstation 20. Originally, the TriggerMan command language was implemented as an extension of Tcl. We have since implemented a command-language parser so Tcl is not a required component of TriggerMan, and commands have a more natural, SQL-like syntax. At the time the tests were done, the createTrigger command was implemented as an extended Tcl command. The performance tests were done in the following manner. A single EMPLOYEE data source was defined. A total of 1350 triggers were then created using the Tcl program shown inset.

```
for {set i 2} {$i <= 2700} {incr i 2} {
  createTrigger t$i in ts1 {
  from EMPLOYEE
  on {insert EMPLOYEE}
  when {EMPLOYEE.salary = [expr 2700 % $i]}
  do {}} # no trigger action was run so only
}          # condition testing would be timed
```

The triggers created fire when the inserted employee has a salary equal to some constant. The [expr 2700 % $i] expression is evaluated before the trigger is created. The % symbol is the modulo operator. There are 24 triggers that will fire when the inserted salary is zero. Triggers are allocated round-robin to the different vprocs. The tests were run first with one vproc, then with two. With one vproc, only one of the processors is used for rule condition testing. With two, both processors are used. The results are summarized in the following table:

Number of Processors Used (number of vprocs).	Average Condition Testing Time for All Triggers	Average Condition Testing Time Per Trigger
1	13.5 msec	10 µsec
2	7.5 msec	5.6 µsec

No selection predicate indexing strategy [Hans90,Hans96b] is currently used. A selection predicate index could dramatically increase performance for this example. This example shows that performance will be quite good for single data source triggers even when their conditions are not or cannot be indexed.

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 being implemented as a vehicle to explore asynchronous trigger processing issues and to validate the design approach introduced here. 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, e.g., 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 (e.g. multiple-table and temporal triggers) 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 asynchronous trigger processing similar to the one described here could be incorporated directly into a DBMS. This would allow the benefits of asynchronous trigger processing, particularly good update response time *plus* sophisticated trigger processing capability, without the need to incur the cost of moving update descriptors across the boundary from the DBMS into another system. In addition, it would not be necessary to cross back to the DBMS to run trigger actions against database data. 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 information monitoring applications more easily, and these applications will run with faster performance.

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.

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

[Chaw96] Chawathe, S., A. Rajaraman , H. Garcia-Molina , and J. Widom, "Change Detection in Hierarchically Structured Information," *Proc. ACM SIGMOD Conf.*, 1996.

[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.

[Hans96c] 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).

[Info97] “Informix Universal Server,” http://www.informix.com

[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.

[Oust94] Ousterhout, John, *Tcl and the Tk Toolkit,* Addison Wesley, 1994.

[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.

[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.