Scalable Trigger Processing

Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha,
Sashi Parthasarathy, J. B. Park and Albert Vernon
301 CSE, CISE Department, University of Florida
Gainesville, FL 32611-6120
hanson@cise.ufl.edu, http://www.cise.ufl.edu/~hanson

Abstract[†]

Current database trigger systems have extremely limited scalability. This paper proposes a way to develop a truly scalable trigger system. Scalability to large numbers of triggers is achieved with a trigger cache to use main memory effectively, and a memory-conserving selection predicate index based on the use of unique expression formats called expression signatures. A key observation is that if a very large number of triggers are created, many will have the same structure, except for the appearance of different constant values. When a trigger is created, tuples are added to special relations created for expression signatures to hold the trigger's constants. These tables can be augmented with a database index or main-memory index structure to serve as a predicate index. The design presented also uses a number of types of concurrency to achieve scalability, including token (tuple)-level, condition-level, rule action-level, and data-level concurrency.

1. Introduction

Trigger features in commercial database products are quite popular with application developers since they allow integrity constraint checking, alerting, and other operations to be performed uniformly across all applications. Unfortunately, effective use of triggers is hampered by the fact that current trigger systems in commercial database products do not scale. Numerous database products only allow one trigger for each type of update event (insert, delete and update) on each table. More advanced commercial trigger systems have effective limits of a few hundred triggers per table.

Application designers could effectively use large numbers of triggers (thousands or even millions) in a single database if it were feasible. The advent of the Internet and the World Wide Web makes it even more important that it be possible to support large numbers of triggers. A web interface could allow users to interactively create triggers over the Internet. This type of architecture could lead to large numbers of triggers created in a single database.

This paper presents strategies for developing a highly scalable trigger system. The concepts introduced here are being implemented in a system we are developing called TriggerMan, which consists of an extension module for an object-relational DBMS (a DataBlade for Informix with Universal Data Option, hereafter simply called Informix [Info99]), plus some additional programs to be described later. The approach we propose for implementing a scalable trigger system uses asynchronous trigger processing and a sophisticated predicate index. This can give good response time for updates, while still allowing processing of large numbers of potentially expensive triggers. The scalability concepts outlined in this paper could also be used in a trigger system inside a DBMS server.

A key concept that can be exploited to develop a scalable trigger system is that if a large number of triggers are created, it is almost certainly the case that many of them have almost the same format. Many triggers may have identical structure except that one constant has been substituted for another, for example. Based on this observation, a trigger system can identify unique *expression signatures*, and group predicates taken from trigger conditions into equivalence classes based on these signatures.

The number of distinct expression signatures is fairly small, small enough that main memory data structures can be created for all of them. In what follows, we discuss the TriggerMan command language and architecture, and then turn to a discussion of how large numbers of triggers can be handled effectively using expression signature equivalence classes and a novel selection predicate indexing technique.

[†] This research was supported by the Defense Advanced Research Projects Agency, NCR Teradata Corporation, and Informix Corporation.

2. The TriggerMan Command Language

Commands in TriggerMan have a keyword-delimited, SQL-like syntax. TriggerMan supports the notion of a *connection* to a local Informix database, a remote database, or a generic data source program. A connection description for a 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 normally correspond to tables, but this is not essential.

Triggers can be defined using this command:

create trigger <triggerName> [in *setName*]
[*optionalFlags*]
from *fromList*
[on *eventSpec*]
[when *condition*]
[group by *attributeList*]
[having *groupCondition*]
do *action*

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 data sources 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 a database. 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 [Hans98].

3. System Architecture

The TriggerMan architecture is made up of the following components:

1. the TriggerMan DataBlade which lives inside of Informix,
2. **data sources**, which normally correspond to local or remote tables. Most commonly, a data source will be a local table. In that case, standard Informix triggers are created automatically by TriggerMan to capture updates to the table. We use the one trigger per table per update event available in Informix to capture updates and transmit them to TriggerMan by inserting

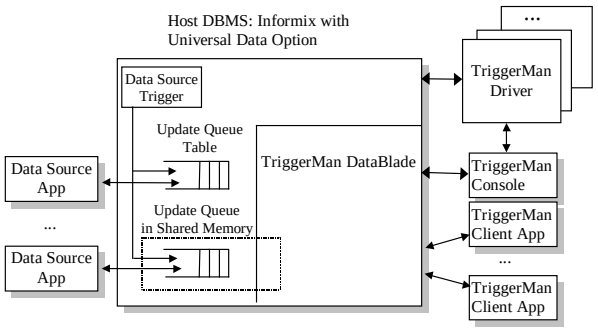

Figure 1. The architecture of the TriggerMan trigger processor.

them in an update descriptor table. For remote data sources, data source applications transmit update descriptors to TriggerMan through the data source API (defined below).

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. 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. Updates received from update capture triggers or data source programs are consumed on the next call to TmanTest().

As Figure 1 shows, data source programs or triggers can place update descriptors in a table acting as a queue. This works in the current implementation. We plan to allow updates to be delivered into a main-memory queue as well in the future. This will deliver updates faster, but the safety of persistent update queuing will be lost. Trigger processing in the current system is asynchronous. If simple Informix triggers are used to capture updates, TriggerMan could process triggers synchronously as well. We plan to add this feature in a later implementation.

TriggerMan is based on an object-relational data model. The current implementation supports char, varchar, integer, and float data types. Support for user-defined types is being added.

Trigger Condition Testing Algorithm

TriggerMan uses a discrimination network called an *A-TREAT network* [Hans96] a variation of the TREAT network [Mira97] for trigger condition testing. In the future, we plan to implement an optimized type of discrimination network called a Gator network in TriggerMan [Hans97b].

This paper focuses primarily on efficient and scalable selection condition testing and rule action execution. The results are applicable to TREAT, Rete [Forg82] and Gator networks when used for trigger condition testing. The results could also be adapted to other trigger systems.

4. General Trigger Condition Structure

Trigger conditions have the following general structure. The **from** clause refers to one or more data sources. The **on** clause may contain an event condition

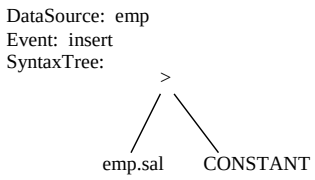

Figure 2. Example expression signature syntax tree.

for at most one of the data sources referred to in the **from** list. The **when** clause of a trigger is a Boolean-valued expression. For a combination of one or more tuples from data sources in the **from** list, the **when** clause evaluates to true or false.

A canonical representation of the **when** clause can be formed in the following way:

1. Translate it to conjunctive normal form (CNF, i.e. and-of-ors notation).
2. Each conjunct refers to zero, one, two, or possibly more data sources. Group the conjuncts by the set of data sources they refer to.

If a group of conjuncts refers to one data source, the logical AND of these conjuncts is a selection predicate. If it refers to two data sources, the AND of its conjuncts is a join predicate. If it refers to zero conjuncts, it is a trivial predicate. If it refers to three or more data sources, we call it a hyper-join predicate.

These predicates may or may not contain constants. The general premise of this paper is that very large numbers of triggers will only be created if predicates in different triggers contain distinct constant values. Below, we will examine how to handle selection and join predicates that contain constants, so that scalability to large numbers of triggers can be achieved.

5. Scalable Predicate Indexing Using Expression Signatures

In what follows, we treat the event (**on**) condition separately from the **when** condition as a convenience. However, event conditions and **when** clause conditions are both logically selection conditions [Hans96] that can be applied to update descriptors submitted to the system.

A *tuple variable* is a symbol, defined in the **from** clause of a trigger, which corresponds to a usage of a particular data source in that trigger. The general form of a selection predicate is:

$(C_{11}\ \text{OR}\ C_{22}\ \text{OR}\ ...\ \text{OR}\ C_{1N_1})\ \text{AND}\ ...\ \text{AND}\ (C_{K1}\ \text{OR}\ C_{K2}\ \text{OR}\ ...\ \text{OR}\ C_{KN_K})$

where all clauses C_{ij} appearing in the predicate refer to the *same* tuple variable. Furthermore, each such clause is an atomic expression that does not contain Boolean operators, other than possibly the NOT operator. A single clause may contain constants.

For convenience, we assume that every data source has a data source ID. A data source corresponds to a single table in a remote or local database, or even a single stream of tuples sent in messages from an application program. An expression signature for a general selection or join predicate expression is a triple consisting of a data source ID, an operation code (insert, delete, update, or insertOrUpdate), and a generalized expression. If a tuple variable appearing in the **from** clause of a trigger does not have any event specified in the **on** clause, then the event is implicitly insert *or* update for that tuple variable. The format of the generalized expression is:

$(C'_{11}\ \text{OR}\ C'_{22}\ \text{OR}\ ...\ \text{OR}\ C'_{1N_1})\ \text{AND}\ ...\ \text{AND}\ (C'_{K1}\ \text{OR}\ C'_{K2}\ \text{OR}\ ...\ \text{OR}\ C'_{KN_K})$

where clause C'_{ij} is the same as C_{ij} except that all constants in C_{ij} are substituted with placeholder symbols. If the entire expression has m constants, they are numbered 1 to m from left to right. If the constant number $x, 1 \le x \le m,$ appears in the clause C_{ij} in the original expression, then it is substituted with placeholder CONSTANT_X in C_{ij} in the expression signature.

As a practical matter, most selection predicates will not contain OR's, and most will have only a single clause. Consider this example trigger condition:

on insert to emp
when emp.salary > 80000

In an implementation, the generalized expression in an expression signature can be a *syntax tree* with placeholders at some leaf nodes representing the location where a constant must appear. For example, the signature of the trigger condition just given can be represented as shown in

Figure **2**. The condition:

on insert to emp
when emp.salary > 50000

has a different constant than the earlier condition, but it has the same signature. In general, an expression signature defines an *equivalence class* of all instantiations of that expression with different constant values.

If an expression is in the equivalence class defined by an expression signature, we say the expression *matches* the expression signature.

Expression signatures represent the logical structure or schema of a part of a trigger condition. We assert that in a real application of a trigger system like TriggerMan, even if very large numbers of triggers are defined, only a relatively small number of unique expression signatures will ever be observed - perhaps a few hundred or a few

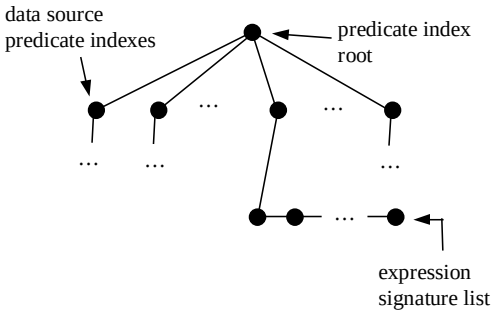

Figure 3. Predicate Index Structure.

thousand at most. Based on this observation, it is feasible to keep a set of data structures in main memory to represent all the distinct expression signatures appearing in all triggers. Since many triggers may have the same signature but contain different constants, tables will be created to store these constants, along with information linking them to their expression signature. When these tables are small, low-overhead main-memory lists or indexes can be used to cache information from them. When they are large, they can be stored as standard tables (with an index when appropriate) and queried as needed, using the SQL query processor, to perform trigger condition testing. We will elaborate further on implementation issues below.

5.1. Processing a Trigger Definition

When a **create trigger** statement is processed, a number of steps must be performed to update the trigger system catalogs and main memory data structures, and to "prime" the trigger to make it ready to run. The primary tables that form the trigger catalogs are these:

trigger_set(tsID, name, comments, creation_date,
isEnabled)

trigger(triggerID, tsID, name, comments, trigger_text,
creation_date, isEnabled, …)

The purpose of the isEnabled field is to indicate whether a trigger or trigger set is currently enabled and eligible to fire if matched by some update. The other fields are self-explanatory. A data structure called the *trigger cache* is maintained in main memory. This contains complete descriptions of a set of recently accessed triggers, including the trigger ID and name,

references to data sources relevant to the trigger, and the syntax tree and Gator network skeleton for the trigger. Given current main memory sizes, thousands of trigger descriptions can be loaded in the trigger cache simultaneously. E.g. if a trigger description takes 4K bytes (a realistic number), and 64Mbytes are allocated to the trigger cache, 16,384 trigger descriptions can be loaded simultaneously.

Another main memory data structure called a *predicate index* is maintained. A diagram of the predicate index is shown in Figure 3. The predicate index can take an update descriptor and identify all predicates that match it.

Expression signatures may contain more than one conjunct. If a predicate has more than one conjunct, a single conjunct is identified as the most selective one. Only this one is indexed directly. If a token matches a conjunct, any remaining conjuncts of the predicate are located and tested against the token. If the remaining clauses match, then the token has completely matched the predicate clause. See [Hans90] for more details on this technique.

The root of the predicate index is linked to a set of *data source predicate indexes* using a hash table on data source ID. Each data source predicate index contains an *expression signature list* with one entry for each unique expression signature that has been used by one or more triggers as a predicate on that data source. For each expression signature that contains one or more constant placeholders, there will be a *constant table.* This is an ordinary database table containing one row for each expression occurring in some trigger that matches the expression signature.

When triggers are created, any new expression signatures detected are added to the following table in the trigger system catalogs:

expression_signature(sigID, dataSrcID, signatureDesc, constTableName, constantSetSize, constantSetOrganization)

The sigID field is a unique ID for a signature. The dataSrcID field identifies the data source on which the signature is defined. The signatureDesc field is a text field with a description of the signature. We will define the other fields later.

When an expression signature E is encountered at trigger creation time, it is broken into two parts: the indexable part, E_I, and the non-indexable part, E_NI, as follows:

E = E_I AND E_NI

The non-indexable portion may be NULL. The format of the constant table for an expression signature containing K distinct constants in its indexable portion is:

const_tableN(exprID, triggerID, nextNetworkNode, const1, … constK, restOfPredicate)

Here, N is the identification number of the expression signature. The fields of const_tableN have the following meaning:

1. exprID is the unique ID of a selection predicate E,
2. triggerID is the unique ID number of the trigger containing E,
3. nextNetworkNode identifies the next A-TREAT network node of trigger triggerID to pass a token to after it matches E (an alpha node or a P-node),
4. const1 … constK are constants found in the indexable portion of E, and
5. restOfPredicate is a description of the non-indexable part of E. The value of restOfPredicate is NULL if the entire predicate is indexable.

If the table is large, and the signature of the indexable part of the predicate is of the form attribute1=CONSTANT1 AND … attributeK=CONSTANTK, the table will have a clustered index on [const1, … constK] as a composite key. If the predicate has a different type of signature based on an operator other than "=", it may still be possible to use an index on the constant fields. As future work, we propose to develop ways to index for non-equality operators and constants whose types are user-defined [Kony98].

Putting a clustered index on the constant attributes will allow the triggerIDs of triggers relevant to a new update descriptor matching a particular set of constant values to be retrieved together quickly without doing random I/O. Notice that const_tableN is not in third normal form. This was done purposely to eliminate the need to perform joins when querying the information represented in the table.

Referring back to the definition of the expression_signature table, we can now define the remaining attributes:

1. constTableName is a string giving the name of the constant table for an expression signature,
2. constantSetSize is the number of distinct constants appearing in expressions with a given signature, and
3. constantSetOrganization describes how the set of constants will be organized in either a main-memory or disk-based structure to allow efficient trigger condition testing. The issue of constant set organization will be covered more fully later in the paper.

Given the disk- and memory-based data structures just described, the steps to process a **create trigger** statement are:

1. Parse the trigger and validate it (check that it is a legal statement).
2. Convert the **when** clause to conjunctive normal form and group the conjuncts by the distinct sets of tuple variables they refer to, as described in section 4.
3. Based on the analysis in the previous step, form a trigger condition graph. This is an undirected graph

with a node for each tuple variable, and an edge for each join predicate identified. The nodes contain a reference to the selection predicate for that node, represented as a CNF expression. The edges each contain a reference to a CNF expression for the join condition associated with that edge. Groups of conjuncts that refer to zero tuple variables or three or more tuple variables are attached to a special "catch all" list associated with the query graph. These will be handled as special cases. Fortunately, they will rarely occur. We will ignore them here to simplify the discussion.

4. Build the A-TREAT network for the rule.
5. For each selection predicate above an alpha node in the network, do the following:
   Check to see if its signature has been seen before by comparing its signature to the signatures in the expression signature list for the data source on which the predicate is defined (see Figure 3). If no predicate with the same signature has been seen before,
   - add the signature of the predicate to the list and update the expresssion_signature catalog table.
   - If the signature has at least one constant placeholder in it, create a constant table for the expression signature.

   If the predicate has one or more constants in it, add one row to the constant table for the expression signature of the predicate.

5.2. Alternative Organization Strategies for Expression Equivalence Classes

For a particular expression signature that contains at least one constant placeholder, there may be one or more expressions in its equivalence class that belong to different triggers. This number could be small or large. To get optimal performance over a wide range of sizes of the equivalence classes of expressions for a particular expression signature, alternative indexing strategies are needed. Main-memory data structures with low overhead are needed when the size of an equivalence class is small. Disk-based structures, including indexed or non-indexed tables, are needed when the size of an equivalence class is large.

The following four ways can be considered to organize the predicates in an expression signature's equivalence class:

1. main memory list
2. main memory index
3. non-indexed database table
4. indexed database table

Strategies 3 and 4 *must* be implemented to make it feasible to process very large numbers of triggers containing predicate expressions with the same signature

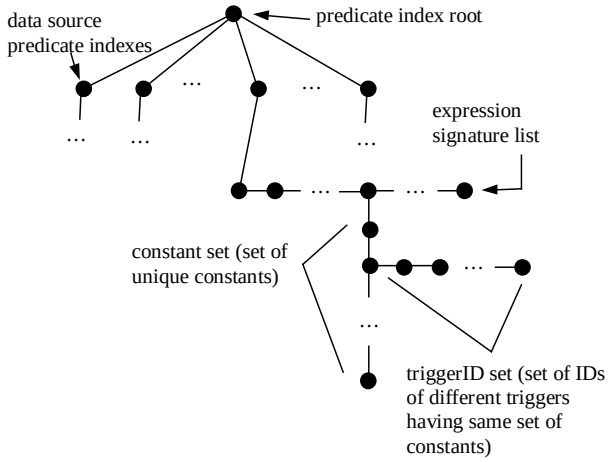

Figure 4. Expanded View of Normalized Predicate Index Structure.

but different constants -- they are mandatory in a scalable trigger system. Strategies 1 and 2 are also required in order to make the common case (a few thousand triggers or less) fast. A cost model that illustrates the tradeoffs is presented in [Hans98b]. Strategies 1 and 2 have been implemented in TriggerMan and strategies 3 and 4 are under construction.

5.3. Common Sub-expression Elimination for Selection Predicates

An important performance enhancement to reduce the total time needed to determine which selection predicates match a token is common sub-expression elimination. This can be achieved by normalizing the predicate index structure. Figure 4 shows an expanded view of the predicate index given in Figure 3. The *constant set* of an expression signature contains one element for each constant (or tuple of constants [const1, … ,constK]) occurring in some selection predicate that matches the signature. Each constant is linked to a triggerID set, which is a set of the ID numbers of triggers containing a particular selection predicate. For example, if there are rules of the form:

create trigger T_I from R when R.a = 100 do …

for I=1 to N, then there will be an expression signature R.a=CONSTANT, the constant set for this signature will contain an entry 100, and the triggerID set for 100 will contain the ID numbers of T_1 … T_N.

We will implement constant sets and triggerID sets in a fully normalized form, as shown in Figure 4, when these sets are stored as either main memory lists or indexes (organizations 1 and 2). This normalized main-memory data structure will be built using the data retrieved from the constant table for the expression signature.

5.4. Processing Update Descriptors Using the Predicate Index

Recall that an update descriptor (token) consists of a data source ID, an operation code, and an old tuple, new tuple, or old/new tuple pair. When a new token arrives, the system passes it to the root of the predicate index, which locates its data source predicate index. For each expression signature in the data source predicate index, a specific type of predicate testing data structure (in-memory list, in-memory lightweight index, non-indexed database table, or indexed database table) is in use for that expression signature. The predicate testing data structure of each of these expression signatures is searched to find matches against the current token.

When a matching constant is found, the triggerID set for the constant contains one or more elements. Each of these elements contains zero or more additional selection predicate clauses. For each element of the triggerID set currently being visited, the additional predicate clause(s) are tested against the token, if there are any.

When a token is found to have matched a complete selection predicate expression that belongs to a trigger, that trigger is *pinned* in the trigger cache. This pin operation is analogous to the pin operation in a traditional buffer pool; it checks to see if the trigger is in memory, and if it is not, it brings it in from the disk-based trigger catalog. The pin operation ensures that the A-TREAT network and the syntax tree of the trigger are in main-memory. After the trigger is pinned, ensuring that it's A-TREAT network is in main memory, the token is passed to the node of the network identified by the nextNetworkNode field of the expression that just matched the token.

Processing of join and temporal conditions is then performed if any are present. Finally, if the trigger condition is satisfied, the trigger action is executed.

6. Concurrent Token Processing and Action Execution

An important way to get better scalability is to use concurrent processing. On an SMP platform, concurrent tasks can execute in parallel. Even on a single processor, use of concurrency can give better throughput and response time by making scarce CPU and I/O resources available to multiple tasks so any eligible task can use them. There are a number of different kinds of concurrency that a trigger system can exploit for improved scalability:

1. **Token-level concurrency:** multiple tokens can be processed in parallel through the selection predicate index and the join condition-testing network.

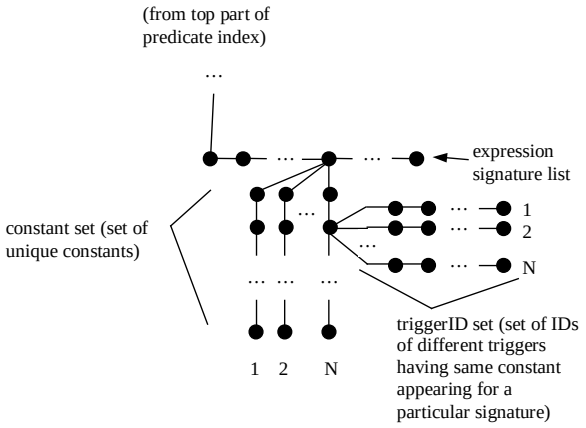

Figure 5. Illustration of partitioned constant sets and triggerID sets to facilitate concurrent processing.

2. **Condition-level concurrency:** multiple selection conditions can be tested against a single token concurrently.
3. **Rule action concurrency:** multiple rule actions that have been fired can be processed at the same time.
4. **Data-level concurrency:** a set of data values in an alpha or beta memory node of an A-TREAT or Gator network [Hans97] can be processed by a query that can run in parallel.

For ideal scalability, a trigger system must be able to capitalize on all four of these types of concurrency. The current implementation supports token level concurrency only. We plan to support the other types of concurrency in future versions of the system. Such a future version will make use of a task queue kept in shared memory to store incoming or internally generated work. An explicit task queue must be maintained because it is not possible to spawn native operating system threads or processes to carry out tasks due to the process architecture of Informix [Info99].

The concurrent processing architecture, as illustrated in **Figure** 1, will make use of N driver processes. We define NUM_CPUS to be the number of real CPUs in the system, and TMAN_CONCURRECY_LEVEL to be the fraction of CPUs to devote to concurrent processing in TriggerMan, which can be in the range (0%,100%]. The TriggerMan administrator can set the TMAN_CONCURRENCY_LEVEL parameter. Its default value is 100%. N is defined as follows:

$N = \lceil NUM\_CPUS * TMAN\_CONCURRENCY\_LEVEL \rceil$

Each driver process will call TriggerMan's TmanTest() function every T time units. Each driver will also call back immediately after one execution of TmanTest() if work is still left to do. We propose a default value of T equal to 250 milliseconds; determining the best value of T is left for future work. TmanTest will do the following:

```
while(total execution time of this invocation of
   TmanTest < THRESHOLD and work is left in the
   task queue)
{
   Get a task from the task queue and execute it.
   Yield the processor so other Informix tasks can use it
   (call the Informix mi_yield routine [Info99]).
}
if task queue is empty
        return TASK_QUEUE_EMPTY
return TASKS_REMAINING
```

The driver program will wait for T time units if the last call to TmanTest() returns TASK_QUEUE_EMPTY. Otherwise, the driver program will immediately call TmanTest() again. The default value of THRESHOLD will be 250 milliseconds also, to keep the task switch overhead between the driver programs and the Informix processes reasonably low, yet avoid a long user-defined routine (UDR) execution. A long execution inside TriggerMan should be avoided since it could result in higher probability of faults such as deadlock or running out of memory. Keeping the execution time inside TriggerMan reasonably short also avoids the problem of excessive lost work if a rollback occurs during trigger processing.

Tasks can be one of the following:

1. process one token to see which rules it matches
2. run one rule action
3. process a token against a set of conditions
4. process a token to run a set of rule actions triggered by that token

Task types 1 and 2 are self-explanatory. Tasks of type 3 and 4 can be generated if conditions and potential actions (triggerID structures containing the "rest of the condition") in the predicate index are partitioned in advance so that multiple predicates can be processed in parallel. An example of when it may be beneficial to partition predicates in advance is when there are many rules with the same condition but different actions. For example, suppose there are M rules of the form:

```
create trigger T_K
from R
when R.company = "IBM"
  do raise event notify_user("user K", R.company,
                             R.sharePrice)
```

for K=1..M. If M is a large number, a speedup can be obtained by partitioning this set of triggers into N sets of equal size. This would result in a predicate index substructure like that illustrated in Figure 5.

Here, the triggerID set would contain references to triggers T_1 … T_M. These references would be partitioned round robin into N subsets of approximately equal size. Multiple subsets would be processed in parallel to achieve a speedup.

7. Trigger Application Design

The trigger system proposed in this paper is designed to be highly scalable. However, just because programmers *can* create a large number of triggers does not mean that is always the best approach. If triggers have extremely regular structure, it may be best to create a single trigger and a table of data referenced in the trigger's **from** clause to customize the trigger's behavior. This is discussed in more detail in a longer version of this paper [Hans98b].

8. Related Work

There has been a large body of work on active database systems, but little of it has focussed on predicate indexing or scalability. Representative works include HiPAC, Ariel, the POSTGRES rule system, the Starburst Rule System, A-RDL, Chimera, RPL, DIPS and Ode [Hans96,McCa89,Ston90,Wido96]. Most active database systems follow the event-condition-action (ECA) model proposed for HiPAC in a straightforward way, testing the condition of every applicable trigger whenever an update event occurs. The cost of this is always at least linear in the number of triggers associated with the relevant event since no predicate indexing is normally used. Moreover, the cost per trigger can be high since checking the condition can involve running an expensive query.

Work by Hanson and Johnson focuses on indexing of range predicates using the interval skip-list data structure [Hans96b], but this approach does not scale to very large numbers of rules since it may use a large amount of main memory. Work on the Rete [Forg82] and TREAT [Mira87] algorithms for efficient implementation of AI production systems is related to the work presented here, but the implicit assumption in AI rule system architectures is that the number of rules is small enough to fit in main memory. Additional work has been done in the AI community on parallel processing of production rule systems [Acha92], but this does not fully address the issue of scaling to large numbers of rules. Issues related to high-performance parallel rule processing in production systems are surveyed by Gupta et al. [Gupt89]. They cite several types of parallelism that can be exploited, including node, intranode, action, and data parallelism. These overlap with the types of concurrency we outlined in section 6. Work by Hellerstein on performing selections after joins in query processing [Hell98] is related to the issue of performing expensive selections after joins in Gator networks and A-TREAT networks [Kand98]. Proper placement of selection predicates in Gator networks can improve trigger system performance, and thus scalability.

The developers of POSTGRES proposed a marking-based predicate indexing scheme, where data and index records are tagged with physical markers to indicate that a rule might apply to them [Ston87,Ston90]. Predicates that can't be solved using an index result in placement of a table-level marker. This scheme has the advantage that the system can determine which rules apply primarily by detecting markers on tables, data records, and index records. Query and update processing algorithms must be extended in minor ways to accomplish this.

A disadvantage of this scheme is that it complicates implementation of storage and index structures. Moreover, when new records are inserted or existing records are updated, a large number of table-level markers may be disturbed. The predicate corresponding to every one of these disturbed markers must be tested against the records, which may be quite time-consuming [Ston87,Ston90]. This phenomenon will occur even for simple predicates of the form attribute=constant if there is no index on the attribute.

Research on the RPL system [Delc88a,Delc88b] addressed the issue of execution of production-rule-style triggers in a relational DBMS, but its developers did not use a discrimination network structure. They instead used an approach that runs database queries to test rule conditions as updates occur. This type of approach has limited scalability due to the potentially large number of queries that could be generated if there are many rules. Work on consistent processing of constraints and triggers in SQL relational databases [Coch96] has helped lead to recent enhancements to the SQL3 standard. However, the focus of this work is on trigger and constraint semantics. An implicit assumption in it is that constraints and triggers will be processed using a query-based approach, which will not scale up to a large number of triggers and constraints. We speculate that it may be possible to work around this assumption. A predicate index like the one proposed in this paper potentially could be used.

The DIPS system [Sell88] uses a set of special relations called COND relations for each condition element (tuple variable) in a rule. These COND relations are queried and updated to perform testing of both selection and join conditions of rules. Embedding *all* selection predicate testing into a process that must query database tables is not particularly efficient – it will not compare favorably to using some sort of main-memory predicate index. A main-memory predicate index should be used to get the best performance for a small-to-medium number of predicates, which is the common case. However, DIPS was capable of utilizing parallelism via the database query processor to test rule conditions, a feature in common with the system described in this paper. The DATEX system addresses the issue of executing large expert systems when working memory is kept in a database [Bran93], and is thus related to rule system scalability. A contribution of the DATEX system was an improved way to represent information normally kept in alpha-memory nodes in TREAT networks. However, DATEX was focussed on large-scale production systems, whereas the work presented in this paper is oriented to handling large numbers of triggers that operate in conjunction with databases and database applications, so our work is not directly comparable to DATEX. In summary, what sets our work apart from prior research efforts on database trigger systems and database-oriented expert systems tools is our focus on scalability from multiple dimensions. These include the capacity to accommodate large numbers of triggers, handle high rates of data update, and efficiently fire large numbers of triggers simultaneously. We achieve scalability through careful selection predicate index design, and support for four types of concurrency (token-level, condition-level, rule-action-level, and data-level).

9. Conclusion

This paper describes an architecture that can be used to build a truly scalable trigger system. As of the date of this writing, this architecture is being implemented as an Informix DataBlade along with a console program, a driver program, and data source programs. The architecture presented is a significant advance over what is currently available in database products. It also generalizes earlier research results on predicate indexing and improves upon their limited scalability [Forg82,Mira87,Ston87,Hans90,Hans96]. This architecture could be implemented in any object-relational DBMS that supports the ability to execute SQL statements inside user-defined routines (SQL callbacks). A variation of this architecture could also be made to work as an external application, communicating with the database via a standard interface (ODBC [Geig95]).

One topic for future research includes developing ways to handle temporal trigger processing [Hans97,AlFa98] in a scalable way, so that large numbers of triggers with temporal conditions can be processed efficiently. Another potential future research topic involves ways to support scalable trigger processing for trigger conditions involving aggregates. Finally, a third potential research topic is to develop a technique to make the implementation of the main-memory and disk-based structures used to organize the constant sets illustrated in Figure 4 extensible, so they will work effectively with new operators and data types. In the end, the results of this paper and the additional research outlined here can make highly efficient, scalable, and extensible trigger processing a reality.

References

[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, August, 1998.

[Bran93] Brant, David A. and Daniel P. Miranker, "Index Support for Rule Activation," *Proceedings of the ACM SIGMOD Conference*, May, 1993, pp. 42-48.

[Coch96] Cochrane, Roberta, Hamid Pirahesh and Nelson Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Systems," *Proceedings of the 22nd VLDB Conference*, pp. 567-578, Bombay, India, 1996.

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

[Delc88a] Delcambre, Lois and James Etheredge, "The Relational Production Language: A Production Language for Relational Databases," *Proceedings of the Second International Conference on Expert Database Systems*, pp. 153-162, April 1988.

[Delc88b] Delcambre, Lois and James Etheredge, "A Self-Controlling Interpreter for the Relational Production Language," *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pp. 396-403, June 1988.

[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.

[Geig95] Geiger, Kyle, *Inside ODBC*, Microsoft Press, 1995.

[Gupt89] Gupta, Anoop, Charles Forgy and Allen Newell, "High Speed Implementations of Rule-Based Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 2, pp. 119-146, May, 1989.

[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, February 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.

[Hans97] Hanson, Eric N., N. Al-Fayoumi, C. Carnes, M. Kandil, H. Liu, M. Lu, J.B. Park, A. Vernon, "TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS," University of Florida CISE Dept. Tech. Report 97-024, December 1997. http://www.cise.ufl.edu.

[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.

[Hans98] Hanson, Eric N., I.C. Chen, R. Dastur, K. Engel, V. Ramaswamy, W. Tan, C. Xu, "A Flexible and Recoverable Client/Server Database Event Notification System," *VLDB Journal*, vol. 7, 1998, pp. 12-24.

[Hans98b] Hanson, Eric N. et al., "Scalable Trigger Processing in TriggerMan," TR-98-008, U. Florida CISE Dept., July 1998. http://www.cise.ufl.edu

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

[Info99] "Informix Dynamic Server, Universal Data Option," http://www.informix.com.

[Kand98] Kandil, Mohktar, *Predicate Placement in Active Database Discrimination Networks*, Ph.D. Dissertation, CISE Department, Univ. of Florida, Gainesville, August 1998.

[Kony98] Konyala, Mohan K., *Predicate Indexing in TriggerMan*, MS thesis, CISE Department, Univ. of Florida, Gainesville, Dec. 1998.

[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.

[Mira87] Miranker, Daniel P., "TREAT A Better Match Algorithm for AI Production Systems," *Proceedings of the AAAI Conference*, August 1987, pp. 42-47.

[Sell88] Sellis, T., C.C. Lin and L. Raschid, "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms," Proceedings *of the 1988 ACM SIGMOD Conference.*

[Ston87] Stonebraker, M., T. Sellis and E. Hanson, "An Analysis of Rule Indexing Implementations in Database Systems," *Expert Database Systems: Proceedings from the First International Workshop*, Benjamin Cummings, 1987, pp. 465-476.

[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.

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