트리거 처리 4 단계 일관성 레벨
(Four Consistency Levels in Trigger Processing)

박 종 범 † Eric Hanson††
(Jongbum Park)

요 약 비동기 트리거 처리기(ATP)는 데이타베이스 트랜잭션의 수행이 완료된 후에 트리거를 처리하는 소프트웨어 시스템이다. ATP 내에서는 트리거 조건의 효율적인 검사를 위하여 차별화 네트워크(discrimination network)가 사용된다. 차별화 네트워크는 내부 상태를 메모리 노드에 저장한다. TriggerMan은 하나의 ATP로써 차별화 네트워크로써 Gator 네트워크를 사용한다.

데이타베이스의 내용 변화는 트리거맨에 토큰 형태로 전달된다. 트리거 조건의 검사는 토큰이 Gator 네트워크를 통과하면서 이루어지는데, 이때 Gator 네트워크의 메모리 노드들이 갱신된다. 토큰의 병렬처리는 시스템의 성능을 향상시키는 여러 방법 중 하나이지만 통제되지 않은 병렬처리는 잘못된 트리거 액션 수행을 유발한다.

이 논문은, 최소한의 이상 현상만을 허용하며 토큰의 병렬 처리를 가능하게 하는, 네 가지 트리거 처리 일관성 레벨을 제안한다. 우리는 각 일관성 레벨에 대하여 병렬 토큰 처리를 가능하게 하는 고유한 기술을 개발하였다. 제안된 기술은 안정된 방법이라는 사실이 공리를 통하여 증명되었으며, 이 기술은 실체화된 (materialized) 뷰 유지 (view maintenance)에 사용될 수 있다.

키워드 : 비동기 트리거 처리기, 엑티브 데이타베이스, 일관성 레벨, 차별화 네트워크, 병렬 토큰 처리, 뷰 유지

Abstract An asynchronous trigger processor (ATP) is a software system that processes triggers after update transactions to databases are complete. In an ATP, discrimination networks are used to check the trigger conditions efficiently. Discrimination networks store their internal states in memory nodes. TriggerMan is an ATP and uses Gator network as the discrimination network.

The changes in databases are delivered to TriggerMan in the form of tokens. Processing tokens against a Gator network updates the memory nodes of the network and checks the condition of a trigger for which the network is built. Parallel token processing is one of the methods that can improve the system performance. However, uncontrolled parallel processing breaks trigger processing semantic consistency.

In this paper, we propose four trigger processing consistency levels that allow parallel token processing with minimal anomalies. For each consistency level, a parallel token processing technique is developed. The techniques are proven to be valid and are also applicable to materialized view maintenance.

Key words : asynchronous trigger processor, active database, consistency levels, discrimination network, parallel token processing, view maintenance

1. Introduction

Active database systems[1,2,3,4] are able to respond automatically to situations of interest that arise in databases. The behavior of an active database is described using triggers[4]. An asynchronous trigger processor (ATP) is a software system that processes triggers (checks trigger conditions and executes trigger actions) after update transactions to databases (data sources, in general) are complete. An ATP,

† 정 회 원 : 공군사관학교 전산통계학과 교수
jbpark@ata.ac.kr
†† 비 회 원 : SQL Server division, Microsoft
ehans@microsoft.com
논문접수 : 2002년 3월 4일
심사완료 : 2002년 9월 24일

TriggerMan, is under development[5,6].

In an ATP, discrimination networks are employed to efficiently check the conditions of the triggers. Discrimination networks were originally developed to check the conditions of the rules in the research on AI production systems[7,8,9,10]. TREAT[11], Rete[12], and Gator[13] are types of discrimination networks. TriggerMan uses Gator (also known as Gator network) as the discrimination network. The changes to the data sources are delivered to TriggerMan in the form of ***tokens***. Parallel token processing is one of the methods that can improve the system performance. First, the topics of trigger, TriggerMan, Gator network, and the necessity of parallel token processing are explained in subsections 1.1, 1.2, 1.3, and 1.4, respectively.

1.1 Trigger

In TriggerMan, a trigger can be defined as follows[14](here, only the clauses relevant to our discussion are shown):

create trigger <trigger_name>
from *from_list*
[**on** *event_spec*]
[**when** *condition*]
do *action*

The *event_spec* can be one of the *insert*, *update*, or *delete* clause. Three tables(*student*, *class*, *enrolment*) that will be used throughout this section and a trigger based on those tables are shown in Figure 1. The trigger *dbms_enrolment* will execute its action when a student enrolls in the DBMS class causing a tuple to be inserted into *enrolment*.

```
Student-schema=(sno, sname, address, phone_umber)
Class-schema=(cno, cname)
Enrolment-schema=(sno, cno)

student (Student-schema)
class (Class-schema)
enrolment (Enrolment-schema)

create trigger dbms_enrolment
from student, class, enrolment
on insert enrolment
when student.sno=enrolment.sno and enrolment.cno=class.cno
and class.cname=DBMS
do print student.sname
```

Figure 1 Three tables and a trigger

1.2 TriggerMan

TriggerMan is an ATP that checks trigger conditions after update transactions to the database are complete. In this paper, database tables and generic data sources are collectively called data sources. In an ATP, trigger actions run outside of data sources. TriggerMan allows a trigger to have a condition based on multiple data sources. TriggerMan uses Gator network as a discrimination network to efficiently test trigger conditions. The internal state of TriggerMan is stored in the host DBMS.

Basically, we assume that tokens are delivered to TriggerMan in serial order. That is, tokens are delivered in the increasing order of timestamps. A token can have one of the following three event types : $\oplus$ (plus; for an insertion operation on data sources), $\ominus$ (minus; for a deletion operation), or δ (delta; for an update operation). A token with event type $\oplus$ or $\ominus$ (: called a token or a $\ominus$ token) contains four components: event type, data source name, tuple, and timestamp. A token with event type δ contains five components: event type, data source name, old tuple, new tuple, and timestamp. Token examples are shown in Figure 2. Depending on the event type, a token contains one or two tuple(s) of a database.

```
(⊕, enrolment, {st500, CS203}, 10:00am)
(⊖, class, {CS101, Introduction to Windows}, 3:00pm)
(δ, student, {st500, JB, Cheongju, 0431-292-1234},
            {st500, JB, Cheongju, 043-292-1234}, 4:32pm)
```

Figure 2. Three token examples

1.3 Gator Network

A Gator network has a general tree structure. The root node of a Gator network called P-node is drawn at the bottom. Other nodes in the Gator network are condition checking nodes and memory nodes. A Gator network is built for each trigger defined in the system. A possible Gator network for the trigger *dbms_enrolment*(Figure 1) is shown in Figure 3. The tokens coming from data sources

are fed into the leaf nodes (condition checking nodes) of the Gator network. The leaf nodes check data source names (table names) from which the tokens are coming.

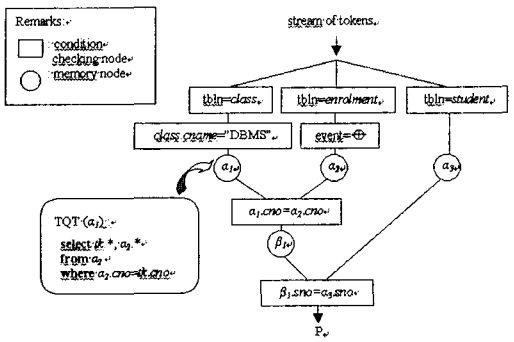

Figure 3 A possible Gator network for *dbms_enrolment*

Memory nodes contain tuples that partially match the condition of a trigger. There are two kinds of memory nodes : **α *memory nodes***(leaf nodes, simply, **α *nodes***) and **β *memory nodes***(internal nodes, simply, **β *nodes***). Every memory node contains the qualifying tuples in it -- so assumed in this paper to make the explanation simple[13].

In Figure 3, an node α_1 logically contains the result of
select * **from** *class* **where** *class.cname*="DBMS"
Similarly, a β node β_1 logically contains the result of
select * **from** *class, enrolment* **where** *class.cname* ="DBMS" **and** *class.cno=enrolment.cno*

The memory nodes of a Gator network are tables in the host DBMS(a commercial database) used for the TriggerMan. The *processing* of a token against a Gator network comprises; (1) the *propagation* of the token downward through the network until no more tokens are propagated or the propagated tokens reach the P-node, (2) the *application* of the token(s) propagated from the token to the memory nodes on the token propagation path, and (3) the *execution* of the action of the trigger, using the tokens arriving at the P-node, for which the Gator network is built. In TriggerMan[15], the application of a token, *tk*, against a node, *n*, depends on the event type of *tk* as follows:

⊕ : Insert the tuple in *tk* into *n*.
⊖ : Delete the tuple in *tk* from *n*.
δ : Delete the old tuple in *tk* from *n*, insert the new tuple in *tk* into *n*.

To propagate tokens in a Gator network, the technique called ***query modification*** is used. When a ⊕ token arrives at a node *n* that has one or more siblings (nodes which feed into the same parent node), the system modifies the ***tuple query template***(TQT) stored in *n*. Each memory node has a TQT in it. The modified query is submitted to the host DBMS to find any tuples matching that token. If the query returns no result, then the propagation stops. Otherwise, tuples in the result are inserted into the parent node *p* of *n*, and the token propagation process is recursively invoked on *p*. For example, suppose a token *tk*, (⊕, *Class*, {CS203, "DBMS"}, 4:45pm), is inserted into α_1 in Figure 3. The system will create the following query by modifying the TQT stored in α_1 using *tk*. In the TQT, by *tk.** we mean all the attribute values of the tuple in *tk*.

select CS203, DBMS, α_2.* **from** α_2
where α_2.*cno*=CS203

If exist, the results are propagated to β_1.

1.4 Necessity of parallel token processing

To improve the performance of TriggerMan, we need to exploit the concurrency available in the system. Processing multiple triggers against a single token is called *trigger-level concurrency*. A study on the implementation of the rule-based systems shows that the true speed-up expected from *rule-level concurrency* is only about two fold [16]. Some of the reasons for this are:

- The average number of rules that need to be processed per change is quite small (around 28 in their example) and is independent on the total number of rules in the system.
- The speed-up obtainable from rule-level parallelism is further reduced by the variance in the processing time of the rules.

A *token* is a kind of *change*, and a *trigger* is a kind of *rule*. However, two properties of Trigger

Man are different from the rule-based system. First, for some applications, the average number of triggers that need to be processed against a token depends on the total number of triggers in the system. Second, the processing of some triggers involves the execution of SQL statements that requires disk access while the processing of others require no disk access (Gator network skeletons are stored in main memory[6]). Therefore, the variance in the trigger processing time can be large.

The first property of TriggerMan(large number of activated triggers per token) could increase the speed-up obtainable from trigger-level concurrency for some applications. The second property of TriggerMan (large variance in trigger processing time) could decrease the speed-up. Since the two properties of TriggerMan have the opposite effect, we expect the speed-up from trigger-level concurrency would be small. Hence, to further improve the performance of TriggerMan, other kinds of concurrencies need to be exploited. They include token-level concurrency, condition-level concurrency, etc.[6,17].

Starting from the observation that uncontrolled parallel token processing breaks trigger processing semantic consistency, this paper introduces an innovative idea for exploiting the token-level concurrency. To process tokens in parallel with controlled and agreed-upon anomalies, we defined four trigger processing consistency levels in Section 2. The proposed technique of parallel token processing is explained in Section 3. Finally, a summary of this paper appears in Section 4.

2. Four Consistency Levels of Trigger Processing

Token-level concurrency means parallel processing of multiple tokens reaching the Trigger Man. However, without concurrency control, the parallel processing of the token that arrive at a Gator network causes various consistency problems. These problems are explained in Section 2.2. To tackle these problems, we define four consistency levels of trigger processing. The definitions of consistency levels appear in Section 2.3. First of all, the notational conventions are summarized in Section 2.1.

2.1 Notational Conventions

The notations used throughout this paper are:

- tk, tk_1, tk_2 : tokens
- tp, tp_1, tp_2 : tuples
- tm, tm_1, tm_2 : timestamps
- $tk_1(\oplus, tp_1)$, $tk_2(\ominus, tp_2, tm_2)$: tokens with event types, tuples, and optional timestamps, data source names are omitted for the simplicity
- $ts(tk_1)$: the timestamp of tk_1
- α, α_1, α_2 : α memory nodes
- β, β_1, β_2 : β memory nodes

2.2 Problems of Uncontrolled Parallel Token Processing

When two tokens, tk_1 and tk_2, consecutively arrive at the Gator network N for a trigger T, assume they are fed into nodes (that can be either an node or a P-node) n_1 and n_2 of N, respectively (see Figure 4). The exhaustive four cases that can happen are:

Case 1. The two nodes are the same ($n_1=n_2$) and tk_1 and tk_2 update a common tuple.

Case 2. The two nodes are the same ($n_1=n_2$) and tk_1 and tk_2 update different tuples.

Case 3. The two nodes are different ($n_1\neq n_2$) and tk_1 and tk_2 or the tokens propagated from them join (have the same join key).

Case 4. The two nodes are different ($n_1\neq n_2$) and tk_1 and tk_2 and the tokens propagated from them do not join.

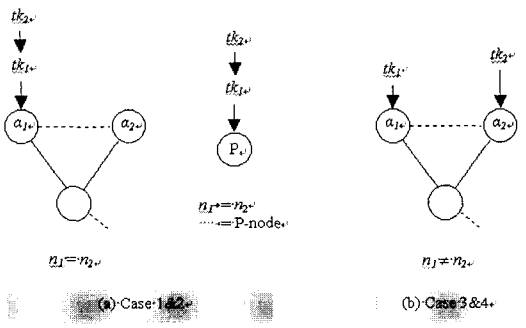

Figure 4 Two tokens arriving at a Gator network

The problems of parallel processing in each case are examined in the following paragraphs.

Case 1 (when $n_1=n_2$ and tk_1 and tk_2 update a common tuple):

An example is given in Figure 5 where tk_1 and tk_2 arrive at an node $\alpha_1(=n_1=n_2)$, consecutively. In parallel execution, if tk_2 is applied to α_1 before tk_1, then later (when tk_1 is processed) tk_1 will remove the new tuple tp_1'. Hence, tp_1' will not exist after two tokens are processed leaving α_1 corrupted. Let us call this ***memory node corruption***.

Assume tk_1 and tk_2 satisfy the condition of the trigger T and n_1 is a P-node of N (Figure 4-a). If tk_1 and tk_2 were processed in serial, the action of T would always be executed using tk_1 first. When they are processed in parallel, the action of T can be executed using tk_2 first which could be unwanted by some users. Let us call this ***out-of-order trigger action execution***.

Case 2 (when $n_1=n_2$ and tk_1 and tk_2 update different tuples):

If $n_1(=n_2)$ is the P-node or both tk_1 and tk_2 are propagated up to the P-node, then the parallel token processing can create the *out-of-order trigger action execution* problem.

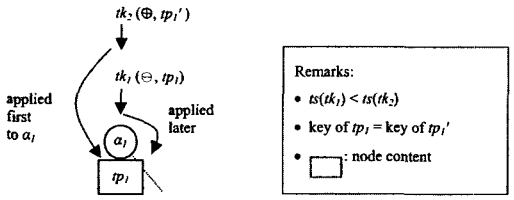

Figure 5 A *memory node corruption*

Case 3 (when $n_1 \neq n_2$ and tk_1 and tk_2 or the tokens propagated from them join):

A part of the Gator network N is shown in Figure 6 where tk_3 deletes tp_3 from α_3 and tk_4 inserts tp_4 into α_4. Assume tk_3 precedes tk_4 and tp_3 and tp_4 join. When two tokens are processed in parallel, if tk_4 is processed before tk_3, then tk_4 (tp_4) will join with tp_3 and propagates tk_4' to β_1. Therefore, a compound tuple $<tp_3, tp_4>$ will be inserted into β_1 and could be used in executing the action of T. However, when following serial processing, tp_3 would be deleted by tk_3 before tk_4 could join it, and tk_4' could not be created. Let us call the compound tuple $<tp_3, tp_4>$ a ***phantom compound tuple***.

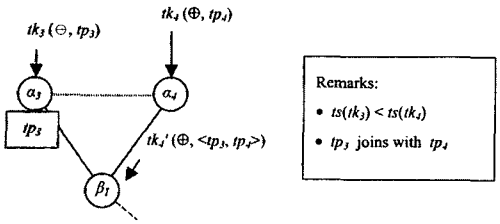

Figure 6 *Untimely joining errors*

In the same circumstances, assume tk_3 inserts (not deletes) tp_3 into α_3 and tk_4 deletes (not inserts) tp_4 from α_4. Following the serial token processing, the compound tuple $<tp_3, tp_4>$ would be inserted into β_1 by tk_3. When the two tokens are processed in parallel, if tk_4 is processed before tk_3, then the compound tuple would not be inserted into β_1. Let us call this the ***lost compound tuple*** problem. The *phantom compound tuple* problem and the *lost compound tuple* problem are collectively called ***untimely joining errors***.

In summary, the two types of *untimely joining errors* are:

- The creation of a compound tuple that never existed (*phantom compound tuple*), because the components of the compound tuple did not exist at the same time period.
- The failure to create a compound tuple that existed for a short time period (*lost compound tuple* problem).

The ***timing error*** is the timestamp difference of two tokens that are involved in the creation of an *untimely joining error*. In Figure 6, when the parallel processing of tk_3 and tk_4 creates a *phantom compound tuple* or a *lost compound tuple* problem, the *timing error* is $ts(tk_4)-ts(tk_3)$.

Case 4 (when $n_1 \neq n_2$ and tk_1 and tk_2 and the tokens propagated from them do not join):

When both tk_1 and tk_2 are propagated to the P-node of the Gator network for T, the parallel processing of the two tokens can create the *out-of-order trigger action execution* problem.

In summary, the uncontrolled parallel processing of tokens that arrive at the Gator network for a trigger T creates the following problems:

- *out-of-order trigger action execution.*
- The action of T is executed using a compound tuple that never existed (*phantom compound tuple*).
- The action of T does not execute because the system cannot detect a (transient) compound tuple (*lost compound tuple*).
- *memory node corruption.*

2.3 Definitions of Trigger Processing Consistency Levels

If we allow *memory node corruption*, then the behavior of TriggerMan would be totally unpredictable. Therefore, *memory node corruption* should never be allowed. A complete list of factors associated with the problems of uncontrolled parallel token processing (Section 2.2) are:

- The existence of the out-of-order trigger action execution problem.
- The existence of an *untimely joining error* in executing a trigger action.
- The amount of timing error when an untimely joining error exists.
- The memory node contents whether they are always true or guaranteed to *stabilize*. (A memory node is said to ***stabilize*** if the parallel application of a set of tokens arriving at the node leaves the same final node content as the content that would be produced by the serial application of the same set of tokens.)

Using these factors, we define four consistency levels in trigger processing. The lower the level is, the higher the performance is, and the higher the level is, the fewer the semantic problems exist. The definitions of consistency levels are:

Level 0 The action of a trigger T will be executed in Level 0 consistency if:

(a) the contents of the memory nodes of the Gator network for T *stabilize*.

Level 1 The action of a trigger T will be executed in Level 1 consistency if:

(a) the contents of the memory nodes of the Gator network for T *stabilize*, and

(b) the *timing error* in an *untimely joining error* is limited to a fixed value.

Level 2 The action of a trigger T will be executed in Level 2 consistency if:

(a) the contents of the memory nodes of the Gator network for T are always correct, and

(b) no *untimely joining error* exists in executing the action of T.

Level 3 The action of a trigger T will be executed in Level 3 consistency if:

(a) the contents of the memory nodes of the Gator network for T are always correct,

(b) no *untimely joining error* exists in executing the action of T, and

(c) no out-of-order trigger action execution problem happens.

3. Parallel Token Processing Technique

Due to the space limitation, among the proposed techniques only the technique for the level 0 consistency is explained in this section. The techniques for other consistency levels appear in [17]. To explain the parallel token processing technique for Level 0 consistency, we need to define some new terms. When tokens are delivered to TriggerMan, the system accumulates them and creates batches. The period during which a batch is formulated is called a ***batch period***. The system processes one batch after another. Tokens in one batch could be processed in parallel. A ***cycle*** is the time during which the tokens in one batch are processed. Assume two tokens processed in a cycle apply to the same tuple (of a memory node), that is, the two tokens have the same key. By ***key*** of a token, we mean the *key of the tuple* in the token. Then, the two tokens are said to be in one ***family***.

When multiple tokens of a family are processed in parallel, the existence and content of the

associated tuple depends on the token processed last. This is a race condition that could corrupt the memory nodes. An example of *memory node corruption* is given in Figure 5 (Section 2.2). Obviously, a corrupted memory node cannot be *stabilized*. Memory node stabilization is a unique condition of Level 0 consistency. Starting from the next paragraph, we explained the technique for memory node stabilization when the tokens arriving at the Gator network are processed in parallel.

In the computer hardware arena, a Translation-Lookaside Buffer (TLB) keeps the recent address translations in order to increase the address translation speed in the virtual memory system[18]. We adopted the idea of TLB and created the ***Stability-Lookaside Buffer*** (SLB) to *stabilize* memory nodes. An SLB maintains information on the tokens which have been applied to a memory node in a cycle. Although it depends on the length of a batch period, the *number of tokens in each family* (***family size***) would be one almost all the time. Still the SLB is needed for the families with size greater than one. The SLB controls the application of tokens in a family for the memory node stabilization. In short, the SLB is different from the TLB in terms of the *theoretical background* (locality: TLB vs. anti-locality: SLB), the *content* (address translation: TLB vs. tuple modification status: SLB), and the *purpose* (increase translation speed: TLB vs. *stabilize* memory nodes: SLB).

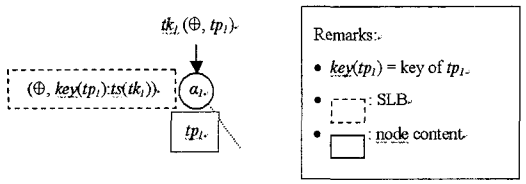

Figure 7 SLB content after a token application

For each tuple (of a memory node) modified (inserted, updated or deleted) during a cycle, three pieces of information (event type, key, timestamp) are stored in an SLB for the memory node. The three-piece information is called a ***line***, and it is about the token that updated a tuple most recently. An SLB is cleared before the beginning of each cycle. An example of an SLB is shown in Figure 7 where tk_1 arrives at a_1. After the application of tk_1 against a_1, the content of the SLB and a_1 are shown in the figure. The SLB maintenance cost can be considered as a tax for the stabilization of memory nodes in the parallel token processing environment.

Let each t_1 and t_2 be a token or an SLB line that have the same key. When the timestamp of t_1 is greater than the timestamp of t_2, we say t_1 is ***younger than*** t_2 or t_2 is ***older than*** t_1. A token that is *younger than* any other token in its family is called the ***youngest*** token.

For the stabilization, each node is equipped with an SLB. Assume a token *tk* arrives at an node, n_1, that has the SLB, B_1. Then, *tk* is applied to n_1 depending on the policy in Figure 8.

- when B_1 does not have a line having the same key as t_k(t_k is the first applied token in its family),
  (i) create a line in B_1 for t_k, and
  (ii) apply t_k to n_1 using Table 1, or
- when B_1 has a line l_1 having the same key as t_k,
  −if t_k is younger than l_1, then
  (i) update l_1 using t_k, and
  (ii) apply t_k to n_1 using Table 1,
  −otherwise, discard t_k since t_k is older than l_1.

Figure 8 Token application policy to a memory node n_1

Theorem 1: If the tokens that arrive at an node are processed using the technique explained in this section (Figure 8), then the node *stabilizes* at the end of each *cycle*.

Proof: Since an α node is a set of tuples, an α node *stabilizes* if and only if each tuple in the node stabilizes. Assume *each tuple* in the node *stabilizes* at the end of each cycle(*condition*). Each tuple in an α node is associated with a (possibly empty) token family in each cycle. Let F be the token family that is associated with an α node tuple tp_1 for a cycle c_1. We can prove the above *condition*

Table 1 The application of a token to a memory node n_l

Case	Token event type	Tuple exists in n_l	Action
1	⊕	No	Apply *tk* as usual (Section 1.3). Propagate *tk*.
2	⊖	Yes	Apply *tk* as usual (Section 1.3). Propagate *tk*.
3	Ä	Yes	Apply *tk* as usual (Section 1.3). Propagate *tk*.
4	⊕	Yes	Transform *tk* into a ä token. (The tuple in n_l may be used as the old tuple of *tk*.) Apply *tk* as usual (Section 1.3). Propagate *tk*.
5	⊖	No	Discard *tk*.
6	Ä	No	Transform *tk* into a ⊕ token using the new tuple in it. Apply *tk* as usual (Section 1.3). Propagate *tk*.

by establishing that the youngest token in F *determines the existence and the content of* (***influences*** in short) tp_l at the end of c_l.

Each token in F will be *applied or considered to be applied* (***considered*** in short) to tp_l during c_l. If the *number of tokens* in F (***size*** of F) is 0 or 1, then obviously tp_l *stabilizes* at the end of c_l. If the size of F is greater than 1, then the influence of the youngest token to tp_l at the end of c_l will be established by proving that the youngest token among the *considered* tokens (in F) *influences* tp_l (*hypothesis*). This will be done using mathematical induction on the number of considered tokens in F.

Basis: When the number of considered tokens is 1, let tk_l be the considered token. Then, tk_l is the youngest token since it is unique and will certainly *influence* tp_l by Figure 8.

Induction: Suppose that the hypothesis is true when the number of considered tokens is $n-1$. Let tk_n be the n-th considered token. Let tk_m be the youngest token among the $n-1$ considered tokens. Then, by the hypothesis, tk_m influences tp_l just before tk_n is considered. If tk_n is *younger than* tk_m, then tk_n (the youngest among the n tokens) will *influence* tp_l by Figure 8. Otherwise (if tk_n is *older than* tk_m), then tk_n will be discarded by Figure 8. In this case, tk_m is the youngest token among the n tokens and will continuously *influence* tp_l. In both cases, the youngest token among the n tokens *influences* tp_l at the end of c_l.

Therefore, the stabilization of a general tuple (tp_l) is established, and it proves the stabilization of an α node. □

The technique for β node stabilization is omitted due to the space limitation.

4. Summary

In this paper, we proposed an innovative idea of performance improvement for TriggerMan. The performance improvement exploits the token-level concurrency. However, as uncontrolled parallel token processing creates problems in trigger processing semantic consistency, four trigger processing consistency levels were defined. The purpose of consistency levels is to increase the system performance on the expense of minimal, controlled, and agreed-upon anomalies. The lower the level is, the higher the performance is, and the higher the level is, the fewer the semantic problems exist.

Level 3 provides serial token processing semantics to the outside world. Level 2 executes trigger actions using each and every consistent data and allows *out-of-order trigger action execution*. Level 1 allows only a limited *timing error* in the data that executes a trigger action. Level 0 guarantees the stabilization of memory nodes. However, in lower consistency levels, a trigger action executes using inconsistent data only when a series of accidents happens. For each consistency level, we developed special techniques for parallel token processing.

To process tokens with full parallelism in Level 0 consistency, the noble technique of SLB was

proposed. The SLB contents are modified by the tokens that are applied to a memory node in a cycle. α node stabilization technique using the SLB was explained in this paper. The technique was proven to be valid. Due to the space limitation, the β node stabilization technique was omitted.

When the proposed techniques are employed in a real system, like TriggerMan, we believe that it will increase the performance of the system considerably while generating consistency problems relatively infrequently. Since a materialized view can be seen as a kind of α node, the proposed techniques can also be used in maintaining materialized views in an innovative way.

Since TriggerMan is under development, the performance improvement analysis of the proposed techniques cannot be executed. Nevertheless, we believe that our technique will improve the system performance in parallel token processing as suggested by other study [16].

References

[1] Dayal, U., Hanson, E. and Widom, J., Active database systems. In W. Kim (Eds.), Modern database systems: the object model, interoperability, and beyond, pp. 434-456, ACM Press, New York, NY, Addison-Wesley, Reading, MA, 1995.

[2] Stonebraker, M., Rowe, L. and Hirohama, M., "The implementation of POSTGRESS," IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.7, pp. 125-142, 1990.

[3] Widom, J., "Starburst active database rule system," IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.4, pp. 583-595, 1996.

[4] Widom, J. and Ceri, S, Introduction to active database systems. In J. Widom & S. Ceri (Eds.), Triggers and Rules for advanced database processing, Morgan Kaufmann, San Francisco, CA, 1996.

[5] Bodagala, S., Optimization of Condition Testing for Multi-Join Triggers in Active Databases, Ph.D. dissertation, CISE dept., Univ. of Florida, 1998.

[6] Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J. and Vernon, A., "Scalable Trigger Processing," Proceedings of the 15th International Conference on Data Engineering, pp. 266-275, Sydney, Australia, 1999.

[7] Acharya, A. and Tambe, M., "Collection-oriented match: Scaling up the data in production systems," (Tech. Report No. CMU-CS-92-218). School of Computer Science, Carnegie Mellon University, 1992.

[8] Butler, P. L., Allen, J. D. and Bouldin, D. W., "Parallel architecture for OPS5," Proceedings of the 15th International Symposium on Computer Architecture, pp. 452-457, 1988.

[9] Gupta, A., Forgy, C., Kalp, D., Newell, A. and Tambe, M., "Result of Parallel implementation of OPS5 on the Encore multiprocessor," (Tech. Report No. CMU-CS-87-146). Computer Science Dept., Carnegie Mellon University, 1988.

[10] Ishida, T., "An optimization algorithm for production systems," IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.4, pp. 549-557, 1994.

[11] Miranker, D. P, TREAT: A new and efficient match algorithm for AI production systems, Morgan Kaufmann, San Mateo, CA, 1990.

[12] Forgy, C. L, "Rete: A fast algorithm for the many pattern/many object pattern match problem," Artificial Intelligence, Vol.19, pp. 17-37, 1982.

[13] Hanson, E. N. and Hasan, M. S, "Gator: An optimized discrimination network for active database rule condition testing,"(Tech. Report No. TR93-036). CISE Dept., University of Florida, 1993.

[14] Cheng, H., Single-table rule condition evaluation in an asynchronous trigger processor, MS thesis, CISE dept., Univ. of Florida, 1997.

[15] Hanson, E. N, Al-Fayoumi, N., Carnes, C., Kandil, M., Liu, H., Lu, M., Park, J. and Vernon, A., "TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS," (Tech. Report No. 97-024). CISE Dept., University of Florida, 1998.

[16] Gupta, A., Forgy, C. and Newell, A., "High-Speed Implementation of Rule-Based Systems," ACM Transactions on Computer Systems, Vol.7, No.2, pp. 119-146, 1989.

[17] Park, J., Parallel Token Processing in an Asynchronous Trigger System, Ph.D. dissertation, CISE dept., Univ. of Florida, 1999.

[18] Patterson, D. A. and Hennessy, J. L, Computer architecture: a quantitative approach, Morgan Kaufmann, San Mateo, CA, 1990.

Jongbum Park

Jongbum Park graduated from the Korea Air Force Academy in 1984. In the field of computer science, he earned a Bachelor's degree at Seoul National University in 1988, a Master's degree at Pohang University of Science and Technology in 1991, and a Ph.D. degree at the University of Florida, in the U.S.A., in 1999. Since then, he has been working as a faculty member at the Korea Air Force Academy. His research areas include Web data searching and active database systems.

Eric N. Hanson

In the field of computer science, Eric Hanson received his B.S. degree from Cornell University in 1983, and received the M.S. and the Ph.D. degrees from the University of California, Berkeley in 1984 and 1987, respectively. From 1992 to 2001, he was an associate professor in the CISE Dept. at the University of Florida. Currently he is with the Microsoft. His research interests include database management and active database systems.